IT IS SLOW

•••

performance analysis and optimisation on the JVM

About

Galo Navarro ~ @srvaroa

- Software engineer: backend, distributed systems, data plumbing
- [...], Last.fm (2008), Tuenti (2011), Midokura (2013), Zhilabs (2016)

Agenda

- Motivation & goals

- Getting answers to:
  - Do I have performance problems?
  - Which one?
  - How can I fix them?

- Takeaways
  - Performance matters
  - Develop with performance in mind
  - Toolbox

Why should I care about performance?

Performance limits business goals

- Resiliency → is your business functioning?
  - DoS: Denial Of Service (or, Die Of Success: Hacker News / Digg / Reddit effect)
- Efficiency
  - Throwing hardware at the problem is not a silver bullet
  - You don’t want “distributed systems” in your problem set
  - Cost is a factor for potential customers (cautionary tale: Scyla DB vs. Cassandra)
  - Not an option in ARM, mobile devices

Why should I care about performance?

Are performance metrics explicit in your business requirements?

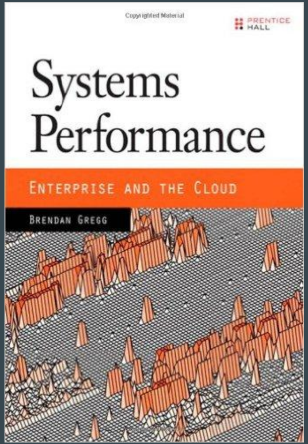
What are your SLAs?

How do latency, throughput.. relate to your business targets?

- Amazon: "it is estimated that a 100-millisecond delay reduces Amazon's sales by 1 percent." [1]
- Google: "..half a second delay caused a 20% drop in traffic"
- Yours?

[1]: http://www.nytimes.com/2009/06/14/magazine/14search-t.html
[2]: http://glinden.blogspot.com.es/2006/11/marissa-mayer-at-web-20.html
[3]: http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/

Performance analysis.. on the JVM



Brendan Gregg: “Systems Performance” (2013). Highlights need for:

- Methodologies (e.g.: USE, TSA..)
- Checklists
- Toolbox

“Linux Performance Analysis in 60.000 ms”

- Protocol for first approach to performance-related incident

Proposes systematising specific approaches for common services (MySQL, Cassandra, Apache..), VMs (Java, Go, Ruby..), etc.

Focusing on low level tools

Profilers

- VisualVM, YourKit, etc. (use instrumentation & JVM Tool Interface)
- Flight Recorder + Mission Control (better: use internal JVM counters & APIs)

Caveats:

- Not always usable on production servers
    - Customers often deny access to their environment
- Focused on dev, forensics, not during incidences or downtime
- Licensing, vendor lock-in (e.g.: Flight Recorder)

MEMORY

Errors

Common originators of an investigation

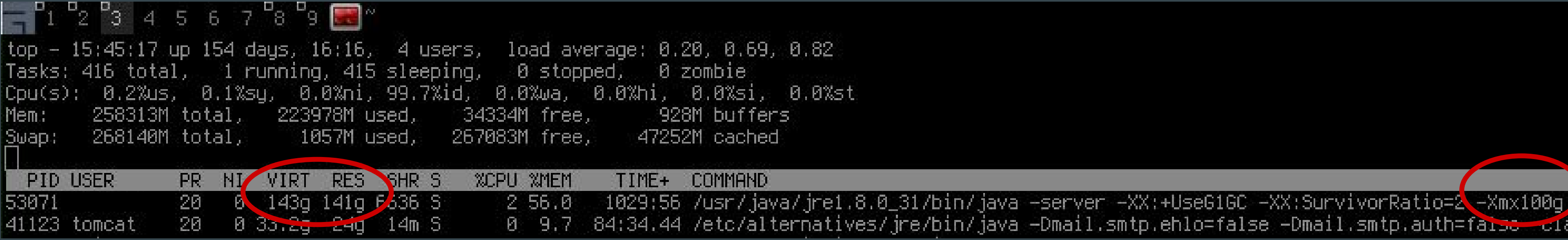
~8 possible reasons

- Out of Memory
  - `java.lang.OutOfMemoryError: Java heap space`
- Too much GC
  - `java.lang.OutOfMemoryError: GC overhead limit exceeded`
- OOM killer (Linux)

  -
    ```
    $ dmesg | grep “Out of memory”
    kernel: Out of memory: Kill process 746 (..) score 1822 or sacrifice child
    ```
- Code Cache
  - `VM warning: CodeCache is full. Compiler has been disabled.`
- `Allocation/Promotion failure, to-space exhausted..`
  - this one is fine: collection required to make room in Eden, Survivor, Old Gen..

Memory footprint

Real memory may be (much) bigger than set by `Xmx`



```
top - 15:45:17 up 154 days, 16:16,  4 users,  load average: 0.20, 0.69, 0.82
Tasks: 416 total,   1 running, 415 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.2%us,  0.1%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    258313M total,   223978M used,    34334M free,      928M buffers
Swap:   268140M total,     1057M used,   267083M free,    47252M cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
53071           20   0  143g 141g 6636 S     2 56.0  1029:56 /usr/java/jre1.8.0_31/bin/java -server -XX:+UseG1GC -XX:SurvivorRatio=2 -Xmx100g
41123 tomcat    20   0 33.2g  24g  14m S     0  9.7 84:34.44 /etc/alternatives/jre/bin/java -Dmail.smtp.ehlo=false -Dmail.smtp.auth=false -cla
```

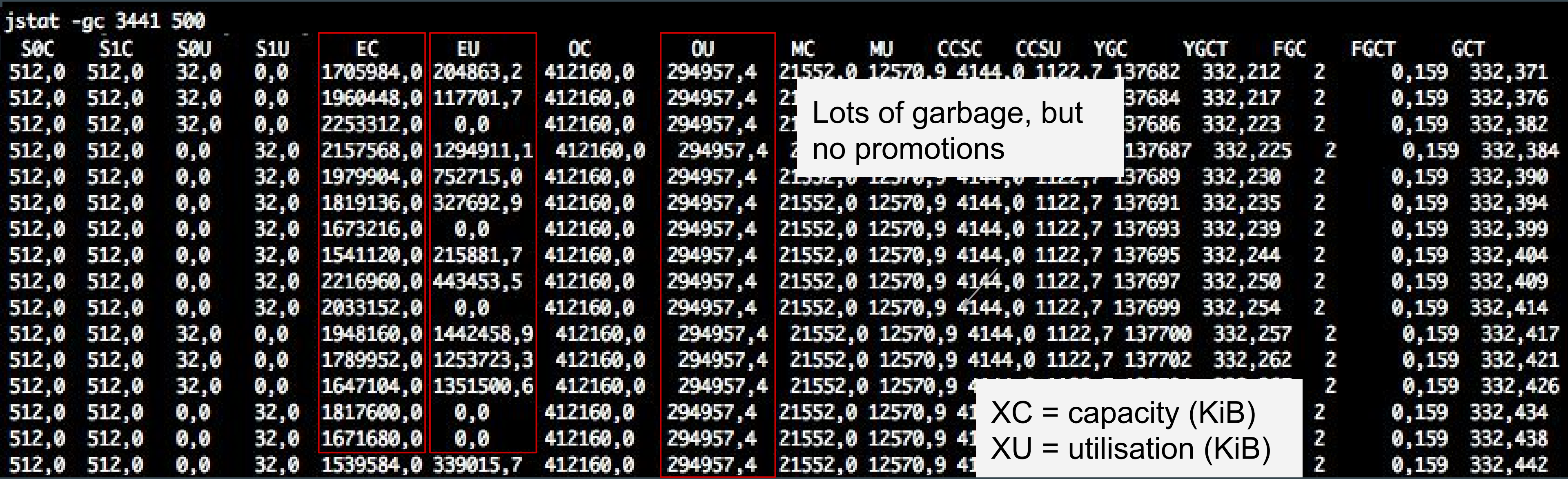
- `top` / `htop` expose real mem usage
- Off-heap:
  - Code cache — Tune with `InitialCodeCacheSize` / `ReservedCodeCacheSize`
  - Off-heap buffers — Tune with `-XX:MaxDirectMemorySize=64M`
  - Thread stacks — Tune with `-Xss=1024k`
- Reclamation: note that batch jobs may hoard memory between runs

Utilisation & Saturation (GC)

```
$ jstat -options
-class
-compiler
-gc
```

```
-gccapacity
-gccause
-gcmetacapacity
-gcold
```

```
-gcoldcapacity
-gcold
-gcoldcapacity
-gcutil
-printcompilation
```



```
jstat -gc 3441 500
 S0C    S1C    S0U    S1U      EC        EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
512,0  512,0  32,0   0,0   1705984,0 204863,2  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137682  332,212   2      0,159  332,371
512,0  512,0  32,0   0,0   1960448,0 117701,7  412160,0   294957,4  21                             37684  332,217   2      0,159  332,376
512,0  512,0  32,0   0,0   2253312,0   0,0     412160,0   294957,4  21                             37686  332,223   2      0,159  332,382
512,0  512,0  0,0   32,0   2157568,0 1294911,1  412160,0   294957,4  2                             137687  332,225   2      0,159  332,384
512,0  512,0  0,0   32,0   1979904,0 752715,0  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137689  332,230   2      0,159  332,390
512,0  512,0  0,0   32,0   1819136,0 327692,9  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137691  332,235   2      0,159  332,394
512,0  512,0  0,0   32,0   1673216,0   0,0     412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137693  332,239   2      0,159  332,399
512,0  512,0  0,0   32,0   1541120,0 215881,7  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137695  332,244   2      0,159  332,404
512,0  512,0  0,0   32,0   2216960,0 443453,5  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137697  332,250   2      0,159  332,409
512,0  512,0  0,0   32,0   2033152,0   0,0     412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137699  332,254   2      0,159  332,414
512,0  512,0  32,0   0,0   1948160,0 1442458,9  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137700  332,257   2      0,159  332,417
512,0  512,0  32,0   0,0   1789952,0 1253723,3  412160,0   294957,4  21552,0 12570,9 4144,0 1122,7 137702  332,262   2      0,159  332,421
512,0  512,0  32,0   0,0   1647104,0 1351500,6  412160,0   294957,4  21552,0 12570,9 4                       2      0,159  332,426
512,0  512,0  0,0   32,0   1817600,0   0,0     412160,0   294957,4  21552,0 12570,9 41                       2      0,159  332,434
512,0  512,0  0,0   32,0   1671680,0   0,0     412160,0   294957,4  21552,0 12570,9 41                       2      0,159  332,438
512,0  512,0  0,0   32,0   1539584,0 339015,7  412160,0   294957,4  21552,0 12570,9 41                       2      0,159  332,442
```

Lots of garbage, but no promotions

XC = capacity (KiB)
XU = utilisation (KiB)

Utilisation & Saturation (GC)

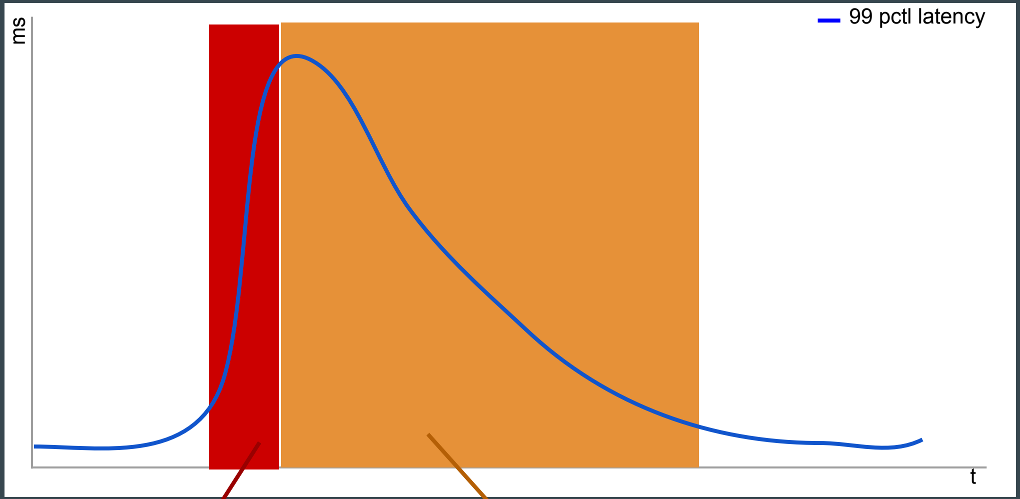
Understanding application pauses and latency spikes

```
-Xloggc:$PATH                         consider ramdisk / ssd [1]
-XX:+PrintGCDetails
-XX:+PrintClassHistogram
-XX:+PrintTenuringDistribution
-XX:+PrintPromotionFailure
-XX:+PrintGCApplicationStoppedTime
-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=$NUM_FILES     default 1
-XX:GCLogFileSize=$SIZE[M|K]          default 512k
-XX:+PrintAdaptiveSizePolicy
```

[1]: https://engineering.linkedin.com/blog/2016/02/eliminating-large-jvm-gc-pauses-caused-by-background-io-traffic

“The app is frozen for 2 seconds..”

Happy case

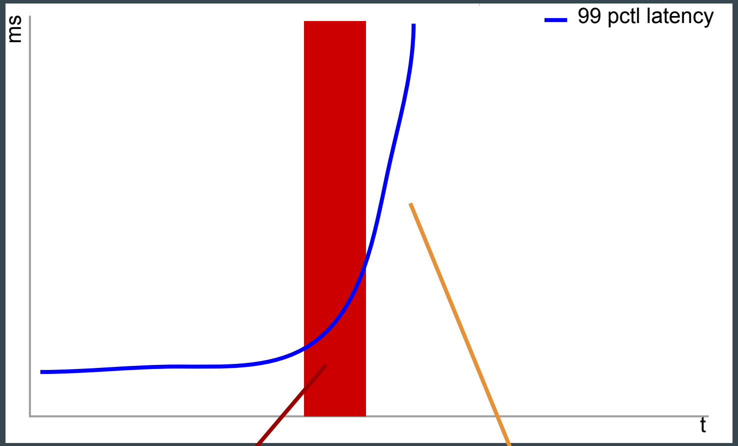


The event itself
requests pile up..

After recovery, system
deals with the queue

“The app is frozen for 2 seconds..”

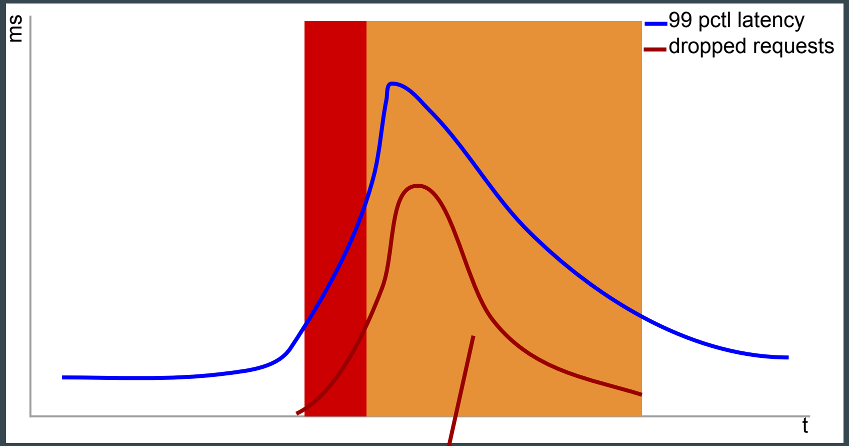
Unhappy case: collapse



the event itself

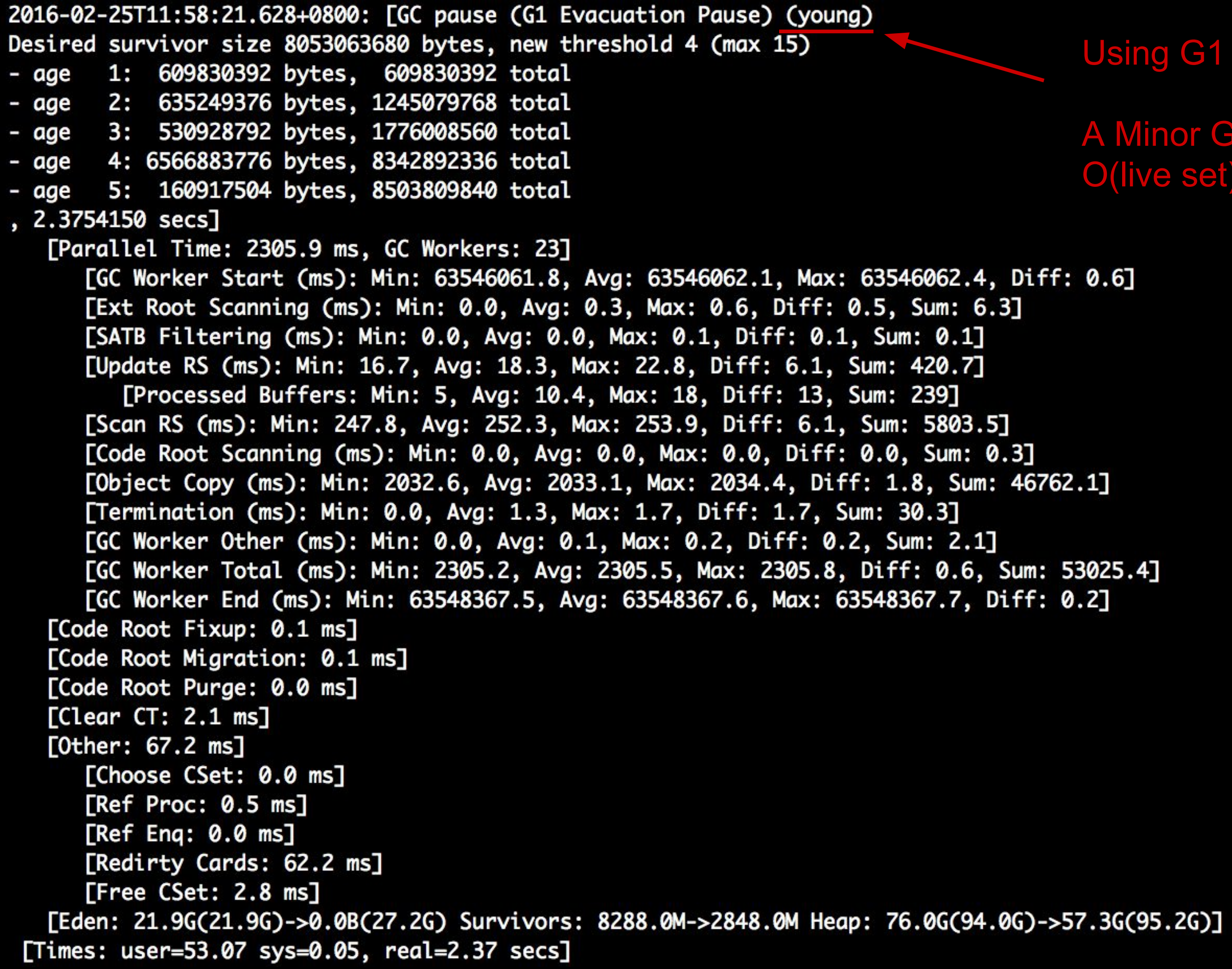
too much backlog , overload

Middle ground: graceful degradation



survive with backpressure

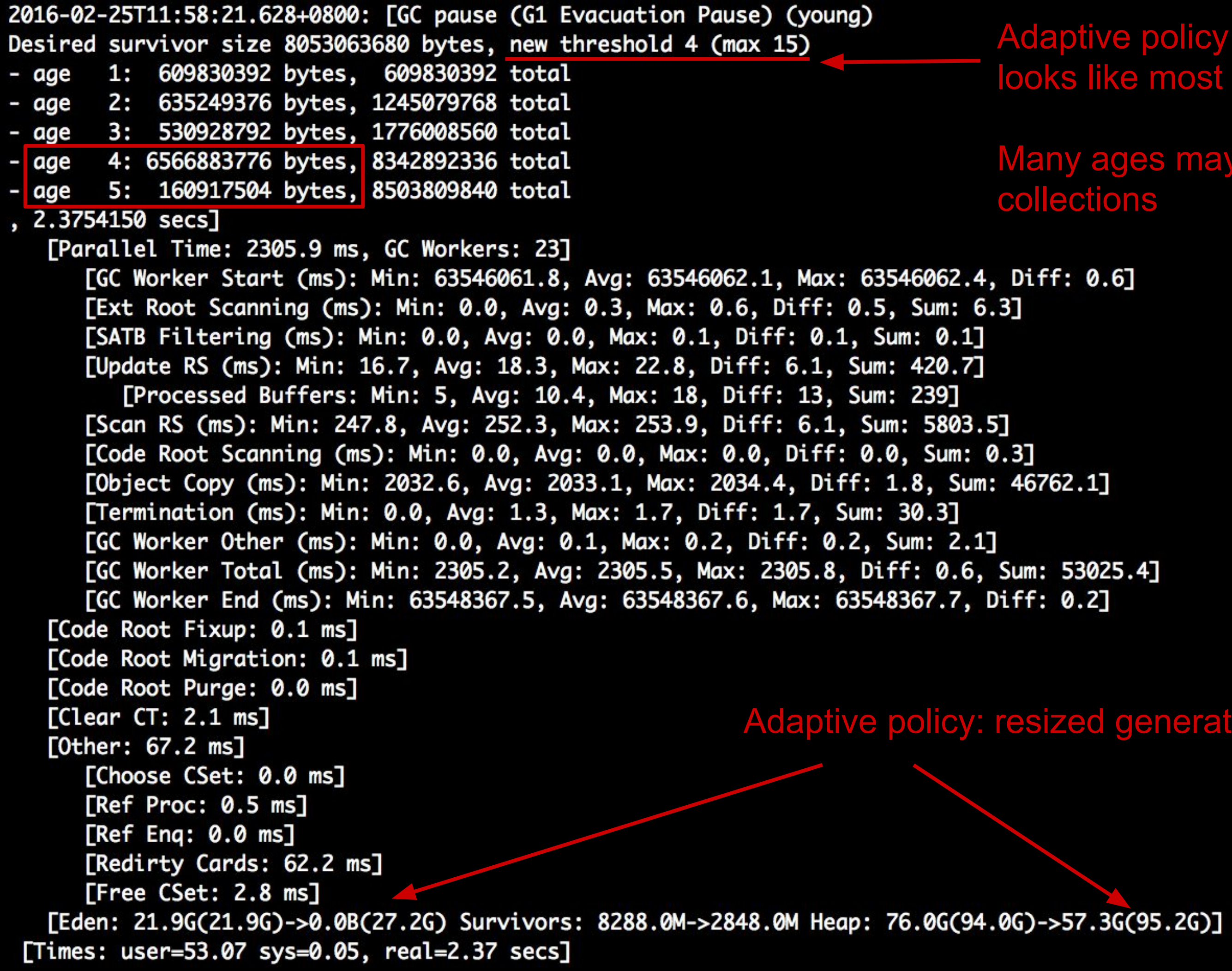
Either way, this service’s clients are not happy



```
2016-02-25T11:58:21.628+0800: [GC pause (G1 Evacuation Pause) (young)
Desired survivor size 8053063680 bytes, new threshold 4 (max 15)
- age   1:  609830392 bytes,  609830392 total
- age   2:  635249376 bytes, 1245079768 total
- age   3:  530928792 bytes, 1776008560 total
- age   4: 6566883776 bytes, 8342892336 total
- age   5:  160917504 bytes, 8503809840 total
, 2.3754150 secs]
   [Parallel Time: 2305.9 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 63546061.8, Avg: 63546062.1, Max: 63546062.4, Diff: 0.6]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.5, Sum: 6.3]
      [SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.1]
      [Update RS (ms): Min: 16.7, Avg: 18.3, Max: 22.8, Diff: 6.1, Sum: 420.7]
         [Processed Buffers: Min: 5, Avg: 10.4, Max: 18, Diff: 13, Sum: 239]
      [Scan RS (ms): Min: 247.8, Avg: 252.3, Max: 253.9, Diff: 6.1, Sum: 5803.5]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 2032.6, Avg: 2033.1, Max: 2034.4, Diff: 1.8, Sum: 46762.1]
      [Termination (ms): Min: 0.0, Avg: 1.3, Max: 1.7, Diff: 1.7, Sum: 30.3]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.1]
      [GC Worker Total (ms): Min: 2305.2, Avg: 2305.5, Max: 2305.8, Diff: 0.6, Sum: 53025.4]
      [GC Worker End (ms): Min: 63548367.5, Avg: 63548367.6, Max: 63548367.7, Diff: 0.2]
   [Code Root Fixup: 0.1 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.1 ms]
   [Other: 67.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.5 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 62.2 ms]
      [Free CSet: 2.8 ms]
   [Eden: 21.9G(21.9G)->0.0B(27.2G) Survivors: 8288.0M->2848.0M Heap: 76.0G(94.0G)->57.3G(95.2G)]
 [Times: user=53.07 sys=0.05, real=2.37 secs]
```

Using G1 (`-XX:+UseG1GC`)

A Minor GC: Stop the World event
O(live set)

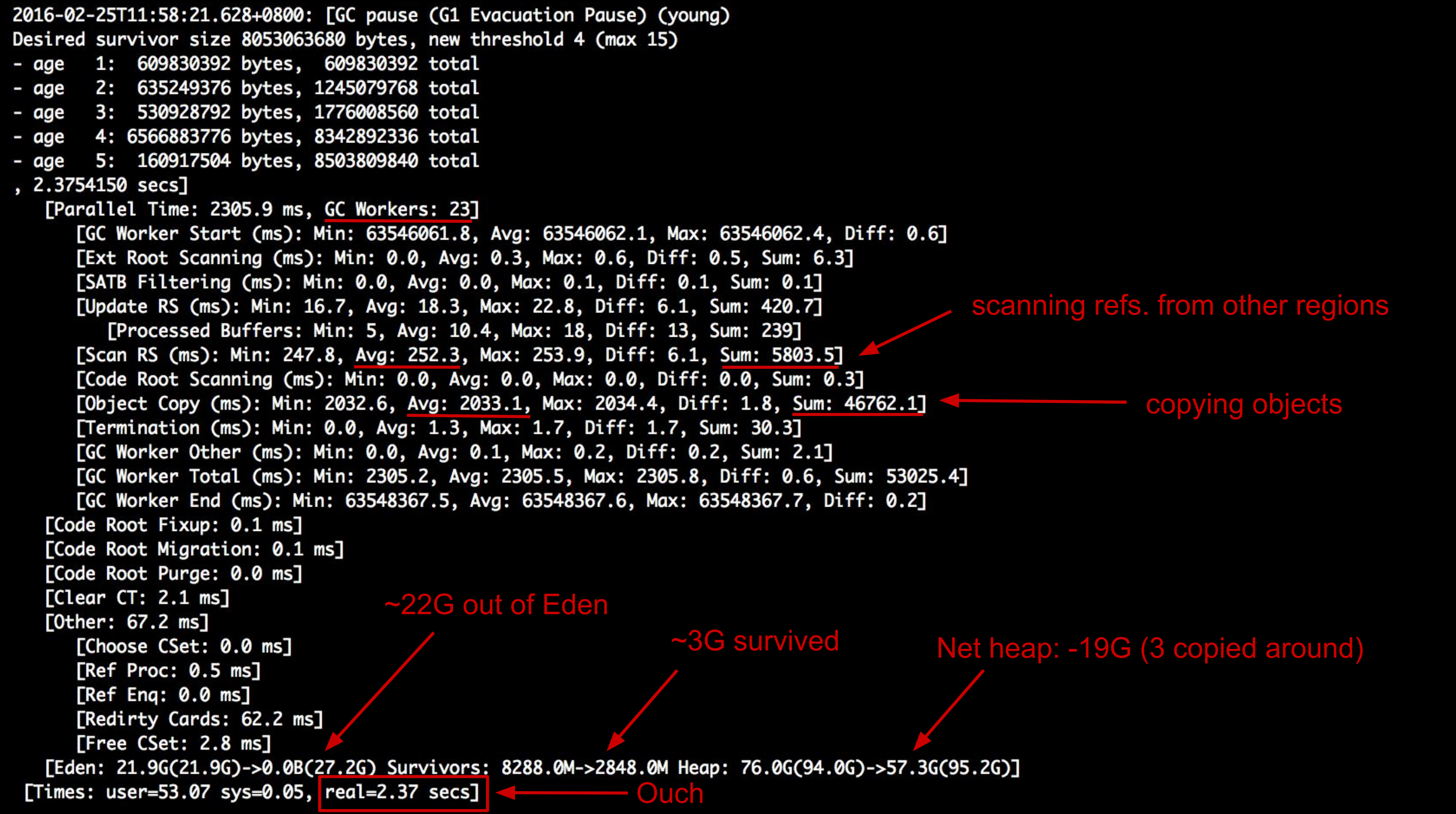


```
2016-02-25T11:58:21.628+0800: [GC pause (G1 Evacuation Pause) (young)
Desired survivor size 8053063680 bytes, new threshold 4 (max 15)
- age   1:  609830392 bytes,  609830392 total
- age   2:  635249376 bytes, 1245079768 total
- age   3:  530928792 bytes, 1776008560 total
- age   4: 6566883776 bytes, 8342892336 total
- age   5:  160917504 bytes, 8503809840 total
, 2.3754150 secs]
   [Parallel Time: 2305.9 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 63546061.8, Avg: 63546062.1, Max: 63546062.4, Diff: 0.6]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.5, Sum: 6.3]
      [SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.1]
      [Update RS (ms): Min: 16.7, Avg: 18.3, Max: 22.8, Diff: 6.1, Sum: 420.7]
         [Processed Buffers: Min: 5, Avg: 10.4, Max: 18, Diff: 13, Sum: 239]
      [Scan RS (ms): Min: 247.8, Avg: 252.3, Max: 253.9, Diff: 6.1, Sum: 5803.5]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 2032.6, Avg: 2033.1, Max: 2034.4, Diff: 1.8, Sum: 46762.1]
      [Termination (ms): Min: 0.0, Avg: 1.3, Max: 1.7, Diff: 1.7, Sum: 30.3]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.1]
      [GC Worker Total (ms): Min: 2305.2, Avg: 2305.5, Max: 2305.8, Diff: 0.6, Sum: 53025.4]
      [GC Worker End (ms): Min: 63548367.5, Avg: 63548367.6, Max: 63548367.7, Diff: 0.2]
   [Code Root Fixup: 0.1 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.1 ms]
   [Other: 67.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.5 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 62.2 ms]
      [Free CSet: 2.8 ms]
   [Eden: 21.9G(21.9G)->0.0B(27.2G) Survivors: 8288.0M->2848.0M Heap: 76.0G(94.0G)->57.3G(95.2G)]
 [Times: user=53.07 sys=0.05, real=2.37 secs]
```

Adaptive policy. Threshold goes 4 → 5
looks like most objects don’t survive past 4

Many ages may suggest too frequent
collections

Adaptive policy: resized generations



```
2016-02-25T11:58:21.628+0800: [GC pause (G1 Evacuation Pause) (young)
Desired survivor size 8053063680 bytes, new threshold 4 (max 15)
- age   1:  609830392 bytes,  609830392 total
- age   2:  635249376 bytes, 1245079768 total
- age   3:  530928792 bytes, 1776008560 total
- age   4: 6566883776 bytes, 8342892336 total
- age   5:  160917504 bytes, 8503809840 total
, 2.3754150 secs]
   [Parallel Time: 2305.9 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 63546061.8, Avg: 63546062.1, Max: 63546062.4, Diff: 0.6]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.5, Sum: 6.3]
      [SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.1]
      [Update RS (ms): Min: 16.7, Avg: 18.3, Max: 22.8, Diff: 6.1, Sum: 420.7]
         [Processed Buffers: Min: 5, Avg: 10.4, Max: 18, Diff: 13, Sum: 239]
      [Scan RS (ms): Min: 247.8, Avg: 252.3, Max: 253.9, Diff: 6.1, Sum: 5803.5]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 2032.6, Avg: 2033.1, Max: 2034.4, Diff: 1.8, Sum: 46762.1]
      [Termination (ms): Min: 0.0, Avg: 1.3, Max: 1.7, Diff: 1.7, Sum: 30.3]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.1]
      [GC Worker Total (ms): Min: 2305.2, Avg: 2305.5, Max: 2305.8, Diff: 0.6, Sum: 53025.4]
      [GC Worker End (ms): Min: 63548367.5, Avg: 63548367.6, Max: 63548367.7, Diff: 0.2]
   [Code Root Fixup: 0.1 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.1 ms]
   [Other: 67.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.5 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 62.2 ms]
      [Free CSet: 2.8 ms]
   [Eden: 21.9G(21.9G)->0.0B(27.2G) Survivors: 8288.0M->2848.0M Heap: 76.0G(94.0G)->57.3G(95.2G)]
 [Times: user=53.07 sys=0.05, real=2.37 secs]
```

scanning refs. from other regions

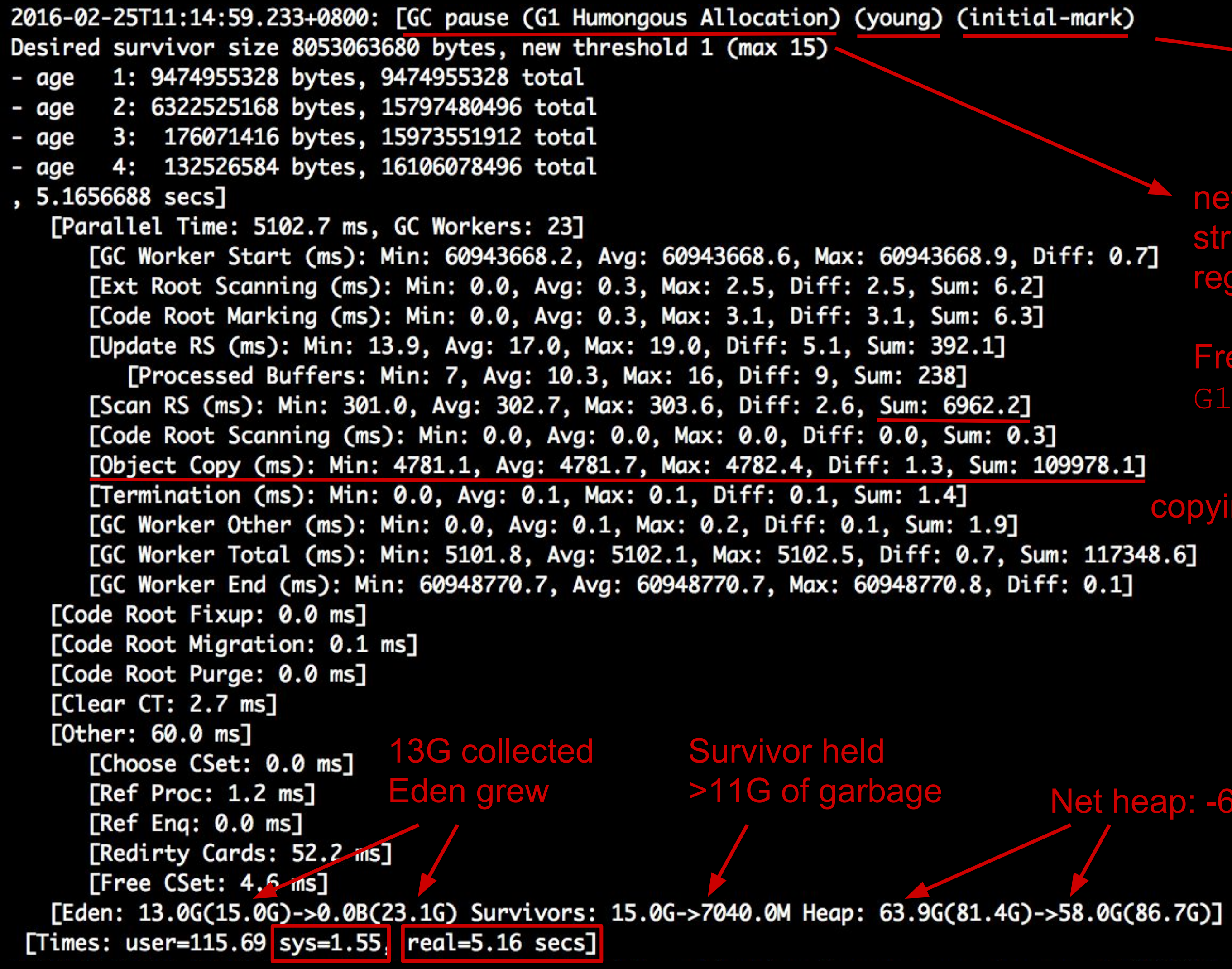
copying objects

~22G out of Eden

~3G survived

Net heap: -19G (3 copied around)

Ouch



```
2016-02-25T11:14:59.233+0800: [GC pause (G1 Humongous Allocation) (young) (initial-mark)
Desired survivor size 8053063680 bytes, new threshold 1 (max 15)
- age   1: 9474955328 bytes, 9474955328 total
- age   2: 6322525168 bytes, 15797480496 total
- age   3:  176071416 bytes, 15973551912 total
- age   4:  132526584 bytes, 16106078496 total
, 5.1656688 secs]
   [Parallel Time: 5102.7 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 60943668.2, Avg: 60943668.6, Max: 60943668.9, Diff: 0.7]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 2.5, Diff: 2.5, Sum: 6.2]
      [Code Root Marking (ms): Min: 0.0, Avg: 0.3, Max: 3.1, Diff: 3.1, Sum: 6.3]
      [Update RS (ms): Min: 13.9, Avg: 17.0, Max: 19.0, Diff: 5.1, Sum: 392.1]
         [Processed Buffers: Min: 7, Avg: 10.3, Max: 16, Diff: 9, Sum: 238]
      [Scan RS (ms): Min: 301.0, Avg: 302.7, Max: 303.6, Diff: 2.6, Sum: 6962.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 4781.1, Avg: 4781.7, Max: 4782.4, Diff: 1.3, Sum: 109978.1]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 1.4]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.1, Sum: 1.9]
      [GC Worker Total (ms): Min: 5101.8, Avg: 5102.1, Max: 5102.5, Diff: 0.7, Sum: 117348.6]
      [GC Worker End (ms): Min: 60948770.7, Avg: 60948770.7, Max: 60948770.8, Diff: 0.1]
   [Code Root Fixup: 0.0 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.7 ms]
   [Other: 60.0 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 1.2 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 52.2 ms]
      [Free CSet: 4.6 ms]
   [Eden: 13.0G(15.0G)->0.0B(23.1G) Survivors: 15.0G->7040.0M Heap: 63.9G(81.4G)->58.0G(86.7G)]
 [Times: user=115.69 sys=1.55, real=5.16 secs]
```

old phase piggy-backing on the young collection

new object > 50% of a region goes straight into OldGen, consume 1 region, and waste part of it

Frequent Humongous? raise `-XX:G1HeapRegionSize`

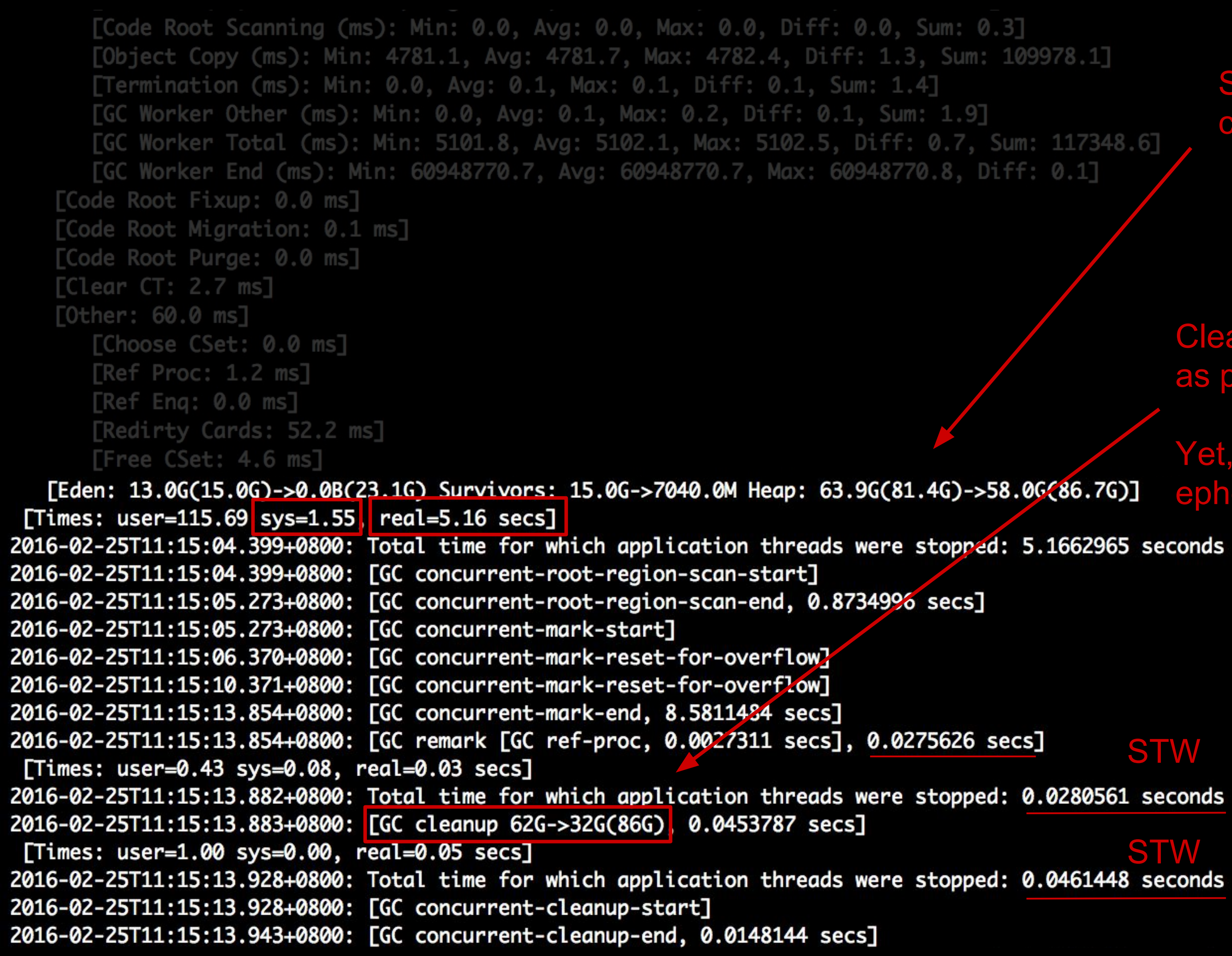
copying stuff around again

13G collected

Eden grew

Survivor held >11G of garbage

Net heap: -6G



```
    [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
    [Object Copy (ms): Min: 4781.1, Avg: 4781.7, Max: 4782.4, Diff: 1.3, Sum: 109978.1]
    [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 1.4]
    [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.1, Sum: 1.9]
    [GC Worker Total (ms): Min: 5101.8, Avg: 5102.1, Max: 5102.5, Diff: 0.7, Sum: 117348.6]
    [GC Worker End (ms): Min: 60948770.7, Avg: 60948770.7, Max: 60948770.8, Diff: 0.1]
  [Code Root Fixup: 0.0 ms]
  [Code Root Migration: 0.1 ms]
  [Code Root Purge: 0.0 ms]
  [Clear CT: 2.7 ms]
  [Other: 60.0 ms]
    [Choose CSet: 0.0 ms]
    [Ref Proc: 1.2 ms]
    [Ref Enq: 0.0 ms]
    [Redirty Cards: 52.2 ms]
    [Free CSet: 4.6 ms]
  [Eden: 13.0G(15.0G)->0.0B(23.1G) Survivors: 15.0G->7040.0M Heap: 63.9G(81.4G)->58.0G(86.7G)]
 [Times: user=115.69 sys=1.55, real=5.16 secs]
2016-02-25T11:15:04.399+0800: Total time for which application threads were stopped: 5.1662965 seconds
2016-02-25T11:15:04.399+0800: [GC concurrent-root-region-scan-start]
2016-02-25T11:15:05.273+0800: [GC concurrent-root-region-scan-end, 0.8734996 secs]
2016-02-25T11:15:05.273+0800: [GC concurrent-mark-start]
2016-02-25T11:15:06.370+0800: [GC concurrent-mark-reset-for-overflow]
2016-02-25T11:15:10.371+0800: [GC concurrent-mark-reset-for-overflow]
2016-02-25T11:15:13.854+0800: [GC concurrent-mark-end, 8.5811484 secs]
2016-02-25T11:15:13.854+0800: [GC remark [GC ref-proc, 0.0027311 secs], 0.0275626 secs]
 [Times: user=0.43 sys=0.08, real=0.03 secs]
2016-02-25T11:15:13.882+0800: Total time for which application threads were stopped: 0.0280561 seconds
2016-02-25T11:15:13.883+0800: [GC cleanup 62G->32G(86G), 0.0453787 secs]
 [Times: user=1.00 sys=0.00, real=0.05 secs]
2016-02-25T11:15:13.928+0800: Total time for which application threads were stopped: 0.0461448 seconds
2016-02-25T11:15:13.928+0800: [GC concurrent-cleanup-start]
2016-02-25T11:15:13.943+0800: [GC concurrent-cleanup-end, 0.0148144 secs]
```

STW ended, continues old collection

Cleanup of 30G not as expensive as previous copying

Yet, we were promoting 30G of ephemeral objects

STW

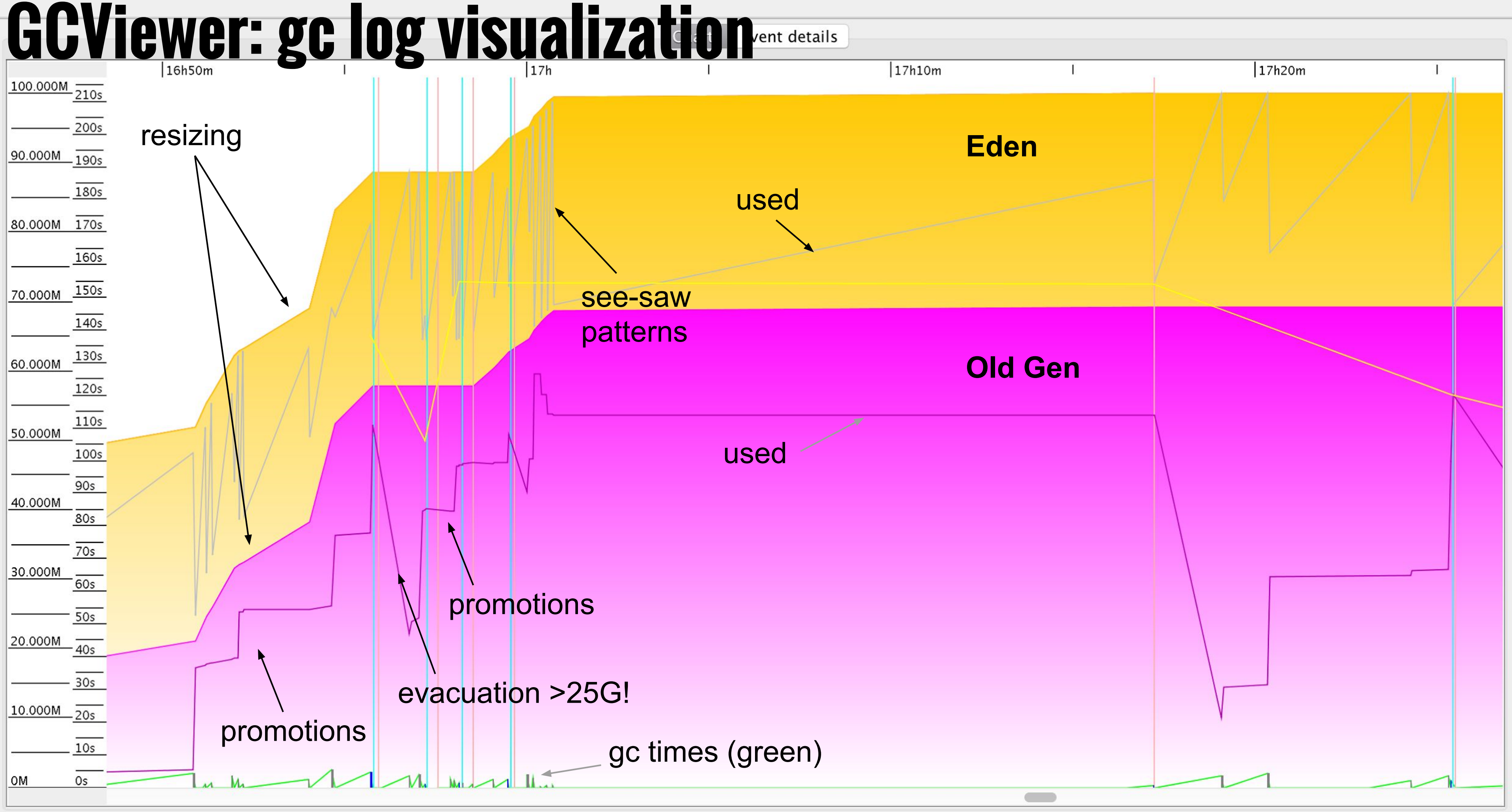
STW

Memory utilisation

App not responsive for > 2s, >6s. What is going on?

- Lots of churn (objects created, soon dereferenced)
  - `jstat`, `jcmd` help seeing at what rate (or Flight Recorder if accessible)
- Allocation pressure → frequent Young GC
  - More copies between survivor spaces
  - Premature promotions create *collection debt*
- Humongous allocations
  - fragmentation → inefficient memory utilisation → even more GC work
- Clients unhappy and/or services downstream cascading failures
  - https://www.elastic.co/blog/elastic-cloud-outage-april-2016
    ZooKeeper (coordination service) dies for GC, major outages in Elastic cloud

GCViewer: gc log visualization



resizing

Eden

used

see-saw patterns

Old Gen

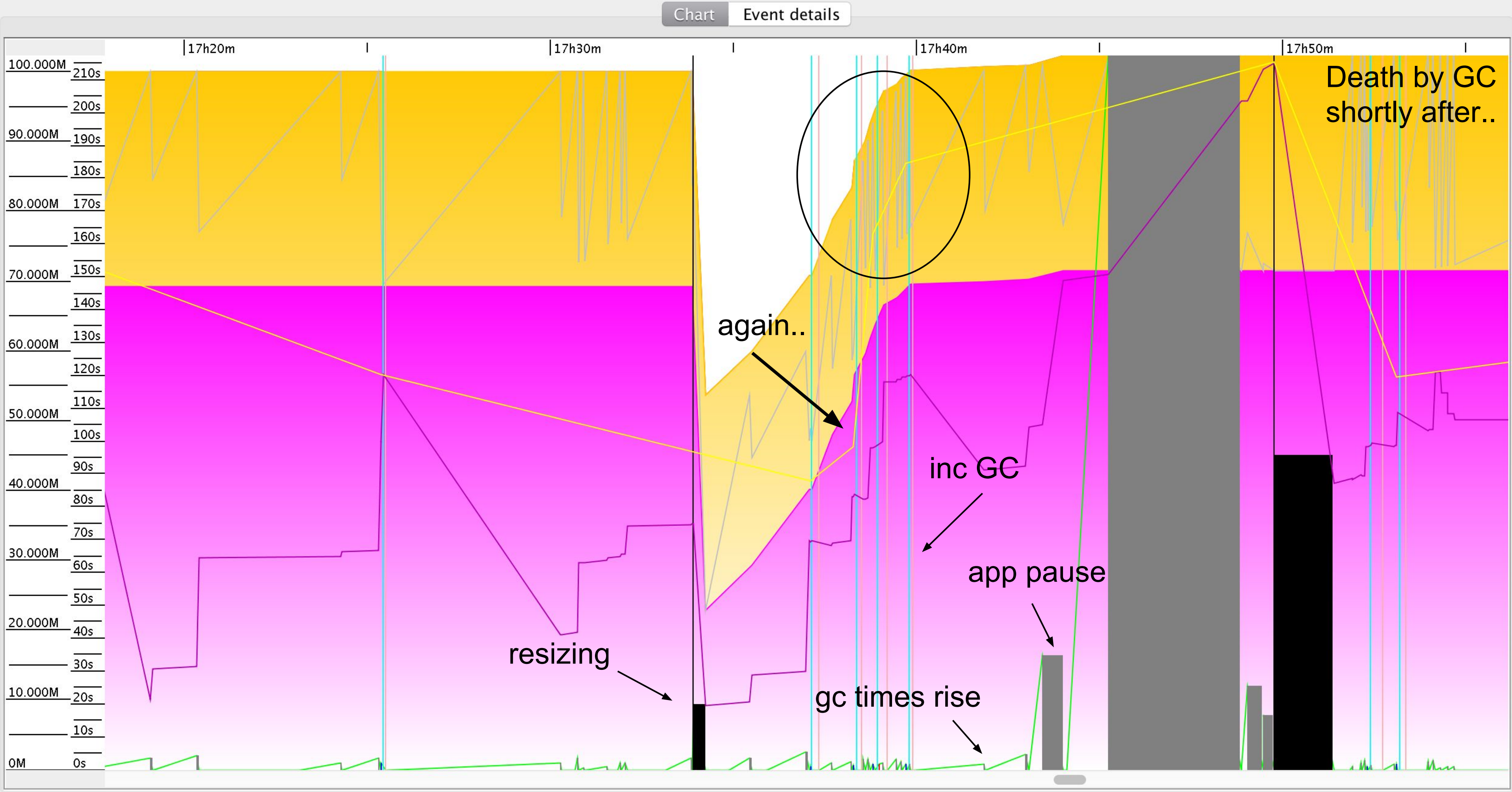
used

promotions

evacuation >25G!

promotions

gc times (green)



Death by GC shortly after..

again..

inc GC

app pause

resizing

gc times rise

GC tuning

Worth several talks in itself. Some knobs that might be relevant..

- `-XX:MaxGCPauseMillis=200` Informs adaptive policies
- `-XX:+PrintReferenceGC` Details into object references
- `-XX:+AlwaysTenure` Straight to Old Gen (spare copies)
- `-XX:+NeverTenure` Never to Old Gen (assume mostly garbage)
- `-XX:+BindGCTaskThreadsToCPUs`
- `-XX:CMSInitiatingOccupancyFraction`
- `-XX:InitiatingHeapOccupancyPercent`Tolerance to utilisation
- `-XX:+ScavengeBeforeFullGC / -XX:+CMSScavengeBeforeRemark`
  Collect eden before Full GC or CMS Remark, sparing the cross-generation ref checks

Application tuning

“The demand upon a resource tends to expand to match its supply.”

~ Parkinson’s Law

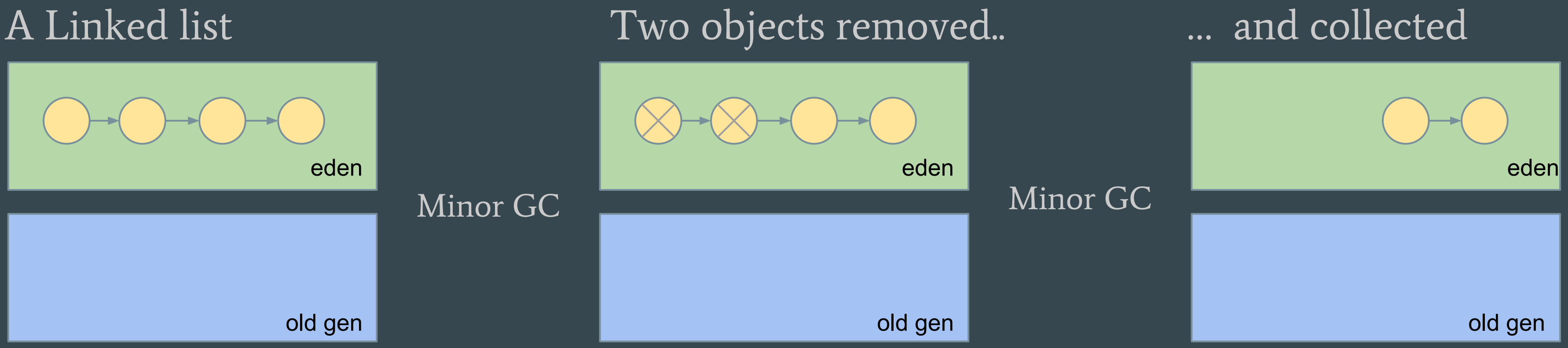
Adding memory generally delays, not fixes, problems.

More space → more garbage → more copies → more collections

Complementary approach

- What (ab)uses these resources? How? Why?

Misbehaviour example: GC nepotism



A Linked list

Minor GC

Two objects removed..

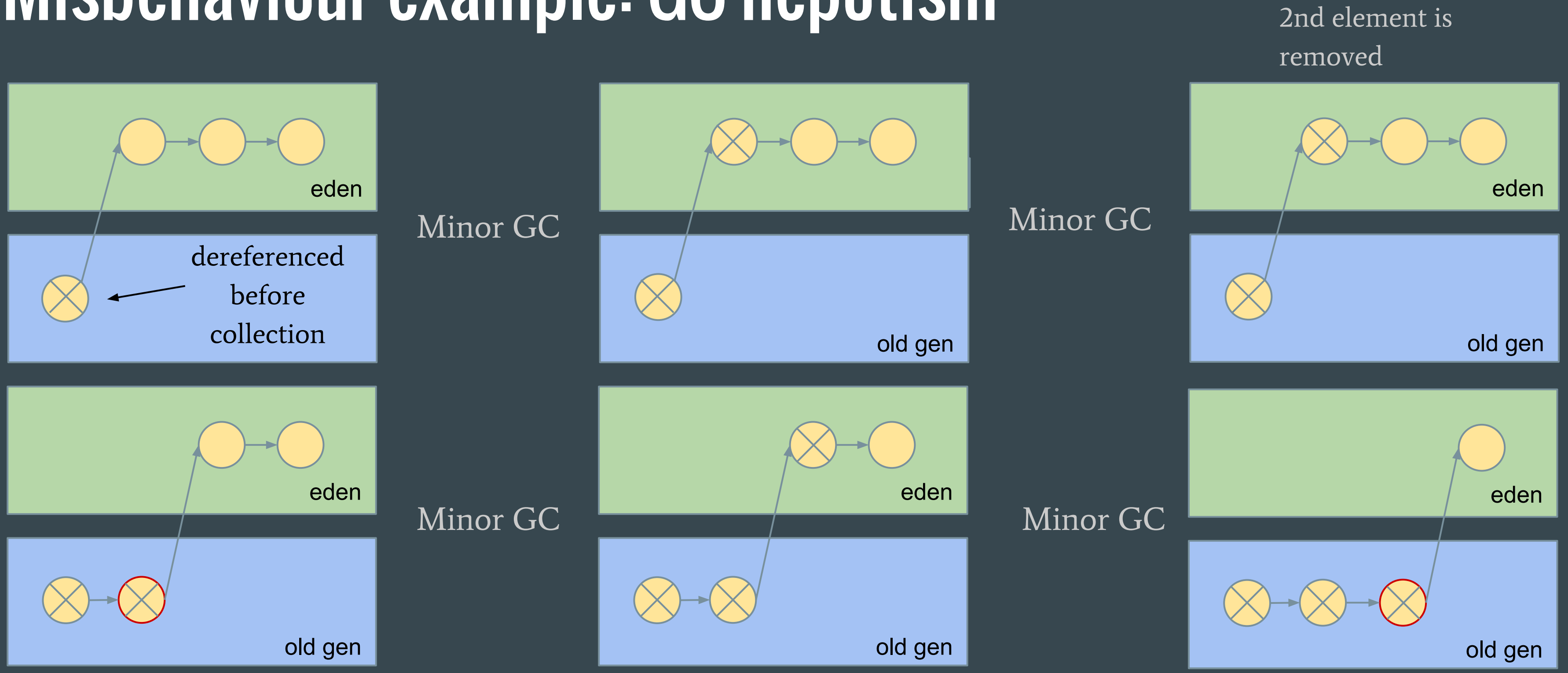
Minor GC

... and collected

All good

Tony Printezis (twitter) https://www.youtube.com/watch?v=M9o1LVfGp2A

Misbehaviour example: GC nepotism



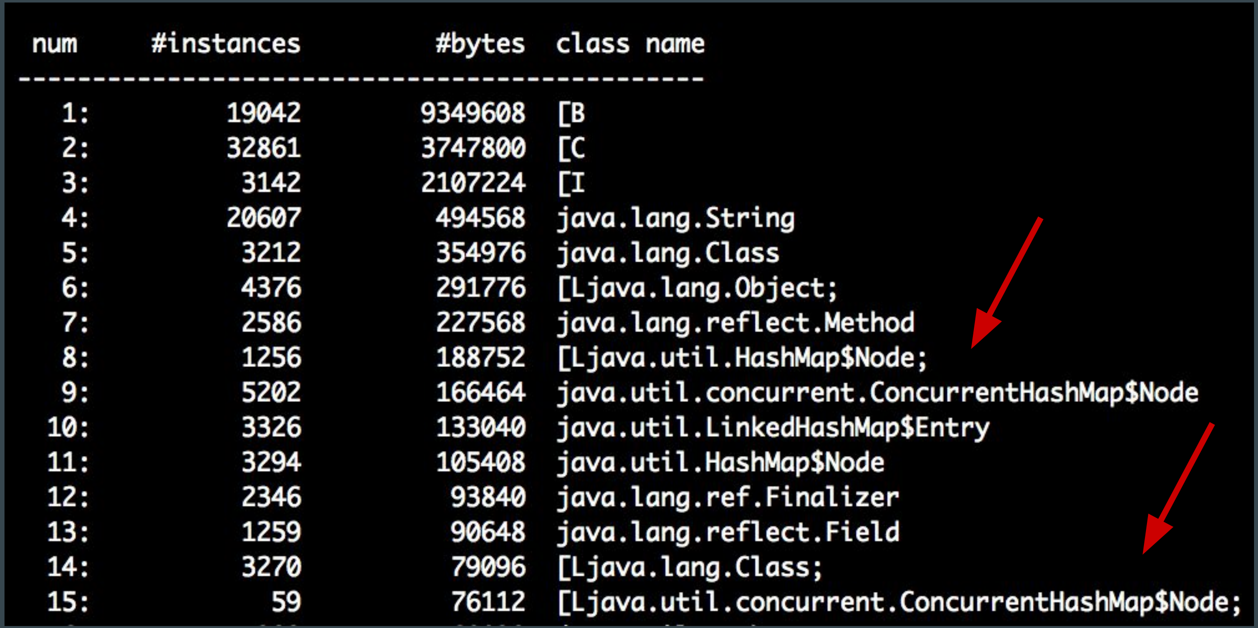
Minor GC doesn't clean old Gen: the deleted object holds a reference. The 2nd survives until promotion.

Tony Printezis (twitter) https://www.youtube.com/watch?v=M9o1LVfGp2A

Measuring memory footprint

```
$ jmap -heap $PID
$ jmap -histo $PID
$ jmap -histo:live $PID
```

Compare with `-histo:live` (full GC, so do it last)



```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         19042        9349608  [B
   2:         32861        3747800  [C
   3:          3142        2107224  [I
   4:         20607         494568  java.lang.String
   5:          3212         354976  java.lang.Class
   6:          4376         291776  [Ljava.lang.Object;
   7:          2586         227568  java.lang.reflect.Method
   8:          1256         188752  [Ljava.util.HashMap$Node;
   9:          5202         166464  java.util.concurrent.ConcurrentHashMap$Node
  10:          3326         133040  java.util.LinkedHashMap$Entry
  11:          3294         105408  java.util.HashMap$Node
  12:          2346          93840  java.lang.ref.Finalizer
  13:          1259          90648  java.lang.reflect.Field
  14:          3270          79096  [Ljava.lang.Class;
  15:            59          76112  [Ljava.util.concurrent.ConcurrentHashMap$Node;
```

[Z = boolean
[B = byte
[S = short
[I = int
[J = long
[F = float
[D = double
[C = char
[L = any non-primitives(Object)

Examining heap

Runtime dump

```
jmap -dump:format=b,file=dump.hprof $PID
jhat dump.hprof
[...]
Started HTTP server on port 7000
```

JVM dump on error

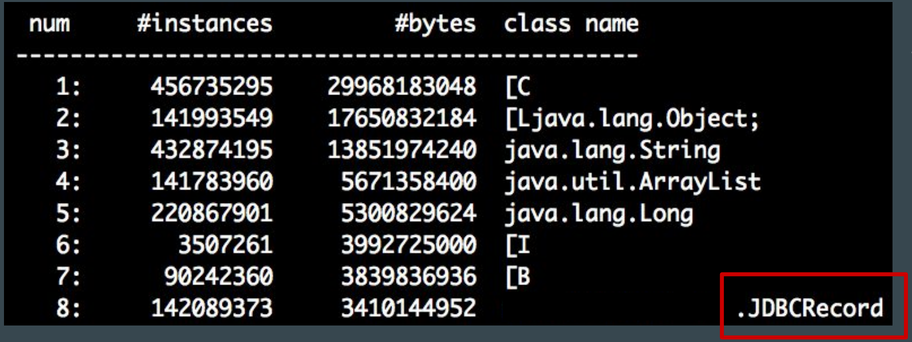
```
-XX:HeapDumpPath=/var/log/my-service/
-XX:+HeapDumpOnOutOfMemoryError
-XX:+HeapDumpAfterFullGC
-XX:+HeapDumpBeforeFullGC
```

Also: Eclipse MAT, Visual VM, Mission Control...

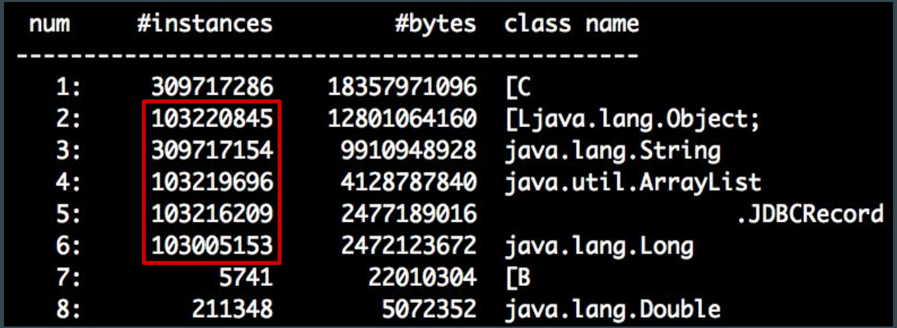
- Lots of candy: track dominator trees, map collisions, object ages, OQL
- Mission control allows defining triggers based on behaviour

Object graph: `http://openjdk.java.net/projects/code-tools/jol/`

Consider the cost of abstractions



```
 num     #instances         #bytes  class name
----------------------------------------------
   1:     456735295    29968183048  [C
   2:     141993549    17650832184  [Ljava.lang.Object;
   3:     432874195    13851974240  java.lang.String
   4:     141783960     5671358400  java.util.ArrayList
   5:     220867901     5300829624  java.lang.Long
   6:       3507261     3992725000  [I
   7:      90242360     3839836936  [B
   8:     142089373     3410144952                  .JDBCRecord
```



```
 num     #instances         #bytes  class name
----------------------------------------------
   1:     309717286    18357971096  [C
   2:     103220845    12801064160  [Ljava.lang.Object;
   3:     309717154     9910948928  java.lang.String
   4:     103219696     4128787840  java.util.ArrayList
   5:     103216209     2477189016                  .JDBCRecord
   6:     103005153     2472123672  java.lang.Long
   7:          5741       22010304  [B
   8:        211348        5072352  java.lang.Double
```

Boxing:

- 5.3G / 220M Long instances = 24 bytes
- Boxing: 3x overhead on long (8 bytes)

Scala (closures, Java conversions, immutables)

Strings:

```
String copy = new String(a + b) // NO
String copy = a + b             // YES
```

`-XX:+PrintStringTableStatistics`

`-XX:+UseStringDeduplication` (only G1)

Consider the cost of abstractions

Object headers

ordinary
object pointer

http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/oop.hpp
http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/markOop.hpp

- 64-bit: 12 bytes padded to multiple of 8 → 16 bytes
- 32-bit: 8 bytes padded to multiple of 4 → 12 bytes

References

- Ref = 4 bytes on < 32G heaps
- Ref = 8 bytes on 64-bit JVMs with >32G heaps

Arrays: 1 ref to type, 4 bytes for length, 1 ref per element. Min 8/16 bytes

Consider the cost of abstractions

- Boxing

```
class A {
      byte x;
}
class B extends A {
      byte y;
}
```

```
new A() = 24 bytes
new B() = 32 bytes
class C {
      Object o = new Object();
}
new C() = 40 bytes..
```

- Growing heap from 24G -> 48G? Think crossing tax brackets..
- `-XX:+UseCompressedOops` will compresses native pointer to 32bits
    - https://wiki.openjdk.java.net/display/HotSpot/CompressedOops
    - Should be enabled in recent JVMs

Consider the cost of abstractions

```
while ((line = reader.readLine()) != null) {
      users.add(new User(line));
}

class User {
      private final String name;
      private final Date birth;
      …
      public User(String s) {
            String[] fields = s.split(“::”);
            this.name = fields[0];
            this.birth = dateFormat.parse(fields[1]);
            ...
      }
      public String getName() { .. }
      public Date getBirth() { return  new Date(birth.getTime) }
      ...
      public String getXXX()
}
```

Good OOP, trying to save CPU on access.. but..

Can we afford multiplying dataset sizes?

Does our internal representation need to mirror the public contract?

Consider the cost of abstractions

```
while ((line = reader.readLine()) != null) {
      users.add(new User(line));
}

class User {
      private final String data;
      …
      public User(String s) {
            this.data = s;
      }
      public String getName() {
            return findField(0);
      }
      public Date getBirth() {
            return new Date(findField(1))
      }
      ...
      private String findField(int n) {
            // loop to find field
      }
```

← Might make sense.. (or, store offsets but not parse) to delay allocation until it’s really needed

- Trades CPU (hardly saturated) for memory
- Think more complex cases:

```
class Ethernet implements L2 {
      MAC src; MAC dst; Short[] vlans;
      L3Packet payload;
}

class IPPacket implements L3 {
      IP src; IP dst; Flags flags;
      L4Datagram payload;
}
```

Unaffordable with millions of instances..

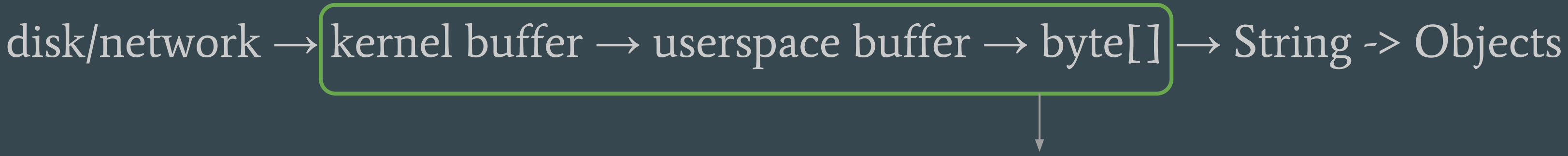
Calculation of object size (only Hotspot)

```
import jdk.nashorn.internal.ir.debug.ObjectSizeCalculator;
import static jdk.nashorn.internal.ir.debug.ObjectSizeCalculator.*

ObjectSizeCalculator sizeCalc = new  ObjectSizeCalculator(
    getEffectiveMemoryLayoutSpecification());

long size = sizeCalc.getObjectSize(new Record(...));
```

Consider the cost of abstractions

Data from/to disk/network into objects implies going through multiple copies..

Typical case:



disk/network → kernel buffer → userspace buffer → byte[ ] → String -> Objects

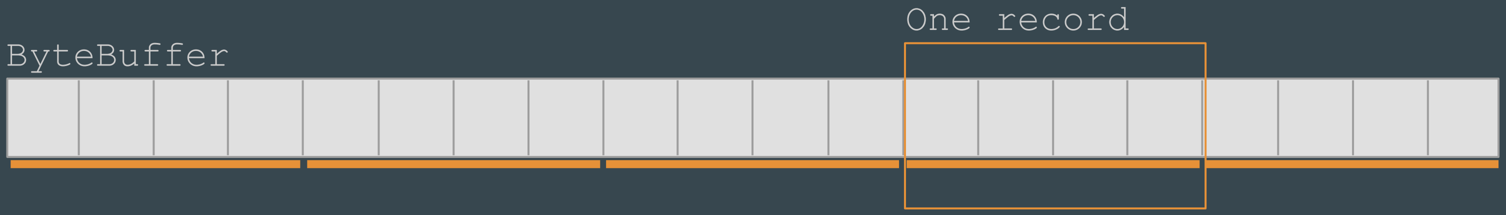
Memory mapped files (useful for large files, IPC..)

```
ByteBuffer b = fileChannel.map(READ_ONLY, 0, file.size())
```

Direct memory buffers (self-managed memory)

```
ByteBuffer dbb = ByteBuffer.allocateDirect(file.size())
fileChannel.read(directByteBuffer)
```

Consider the cost of abstractions



```
ByteBuffer data = ...;
...
while (data.hasRemaining()) {
  ByteBuffer bytes = data.slice()
  bytes.limit(RECORD_SIZE)
  data.position(data.position() + RECORD_SIZE)
  users.add(new Record(bytes))
}
```

- Easy to build an `Iterator[Record]` over a `ByteBuffer`
- Single copy of the data
- Easier to achieve cache friendliness

Examining off-heap memory

Tracking native allocations

JVM flag required

```
    -XX:NativeMemoryTracking=off| summary|detail
```

Retrieve info

```
    $ jcmd $PID VM.native_memory baseline              # set
    $ jcmd $PID VM.native_memory summary.diff          # poll for diff
```

Unsuspected memory sinks

Unsuspected memory sinks lurk everywhere... know your APIs, libraries

- Logs ~ `log.debug("Request " + req.id + " is generating useless garbage")`
  - Strings, Message objects, locks . . .
- Lazy Initialization in Scala: additional int, + sync overhead
- `ArrayList.addAll` → allocates an `Object[size]`
- An object with a `finalize()` method allocates an additional object
  - You don't want finalize() on classes with millions of instances
  - Takes 2 GC cycles to clean
- `WeakHashMap` has a delay to clean dead refs (lazy eviction)
- Secret NIO `ByteBuffer` cache avoids expensive malloc / free sequences for short lived buffers... by potentially caching massive buffers
  - http://mail.openjdk.java.net/pipermail/nio-dev/2015-December/003420.html

CPU

JIT optimisations: Escape analysis + Inlining

```
public A {
  private final int x;
  public A(final int _x) {
     this.x = _x;
     this.y = _y;
  }
  public int getY() { return y; }
}

public void f(int n) {
  int x = 0;
  for (i = 0; i < n; i++) {
     A a = new A(i);
     System.out.println(a.getX());
  }
}
```

(likely) JIT’ed version

```
public void f(int n) {
  int x = 0;
  for (i = 0; i < n; i++) {
     int _x = x;
     System.out.println(_x);
  }
}
```

Objects that don’t escape curr. method or thread might get stack allocation.

Methods calls may get inlined

https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis

Consider the cost of abstractions

- JIT vs OOP: Megamorphic methods can’t be optimized

https://github.com/google/guava/issues/1268

“... guava Immutable collections [...] have specializations for zero (EmptyImmutableList) and one (SingletonImmutableList) element collections. These specializations take the form of subclasses of ImmutableList, to go along with the "Regular" implementation and a few other specializations like ReverseImmutable, SubList, etc.

The result is that when these subclasses mix at some call site, the call is megamorphic, and performance is awful compared to classes without these specializations (worse by a factor of 20 or more).”

JIT: Escape analysis + Inlining

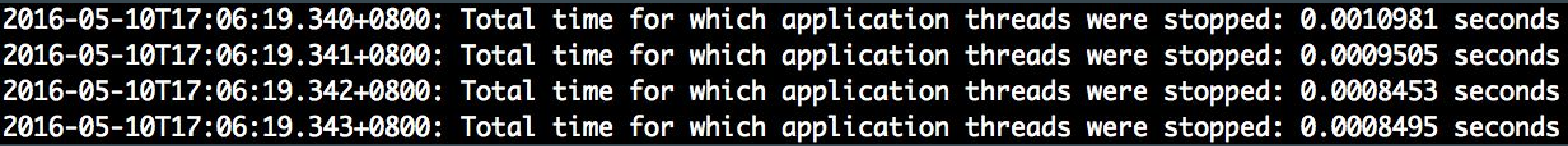
Very relevant for Scala, Java8 lambdas

```
def maybeDouble(Option[Int] o): Option[Long] = {
    o.map { _ * 2 } // o.map( new Function( x: Int) { return x * 2; })
}
```

Help the JIT help you

- Small functions, clean code, immutability, few conditionals, avoid megamorphism
- Profile allocations & benchmark performance to validate assumptions
- JIT watch: https://github.com/AdoptOpenJDK/jitwatch
- Gil Tene: http://infoq.com/presentations/java-jit-optimization

Latency jitter & spikes



```
2016-05-10T17:06:19.340+0800: Total time for which application threads were stopped: 0.0010981 seconds
2016-05-10T17:06:19.341+0800: Total time for which application threads were stopped: 0.0009505 seconds
2016-05-10T17:06:19.342+0800: Total time for which application threads were stopped: 0.0008453 seconds
2016-05-10T17:06:19.343+0800: Total time for which application threads were stopped: 0.0008495 seconds
```

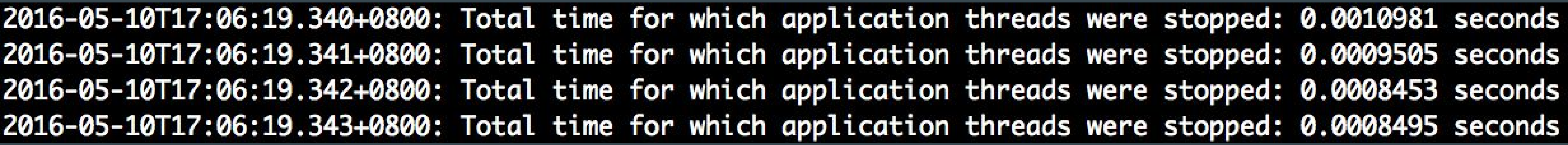
These are not necessarily due to GC. More info using:

```
-XX:+UnlockDiagnosticVMOptions -XX:+PrintSafepointStatistics
```

Time to safepoint. Identified by: `12.754: no vm operation`

- Some JVMs introduce a periodic guaranteed safepoint time (used to perform GC and other tasks, e.g.: apply/revoke code optimisations)
- Can be controlled with `-XX:GuaranteedSafepointInterval=300000`
- http://epickrram.blogspot.com.es/2015/08/jvm-guaranteed-safepoints.html

Latency jitter & spikes



```
2016-05-10T17:06:19.340+0800: Total time for which application threads were stopped: 0.0010981 seconds
2016-05-10T17:06:19.341+0800: Total time for which application threads were stopped: 0.0009505 seconds
2016-05-10T17:06:19.342+0800: Total time for which application threads were stopped: 0.0008453 seconds
2016-05-10T17:06:19.343+0800: Total time for which application threads were stopped: 0.0008495 seconds
```

These are not necessarily due to GC. More info using:

```
    -XX:+UnlockDiagnosticVMOptions -XX:+PrintSafepointStatistics
```

Biased locking. Identified by: `26.319: RevokeBias..`

Optimizes contended locks: last accessor thread has higher chances on next attempt Pro: cache friendliness ; Con: bookkeeping, bad on thread pools, highly concurrent apps..

- Disable with `-XX:-UseBiasedLocking`

Stack dumps

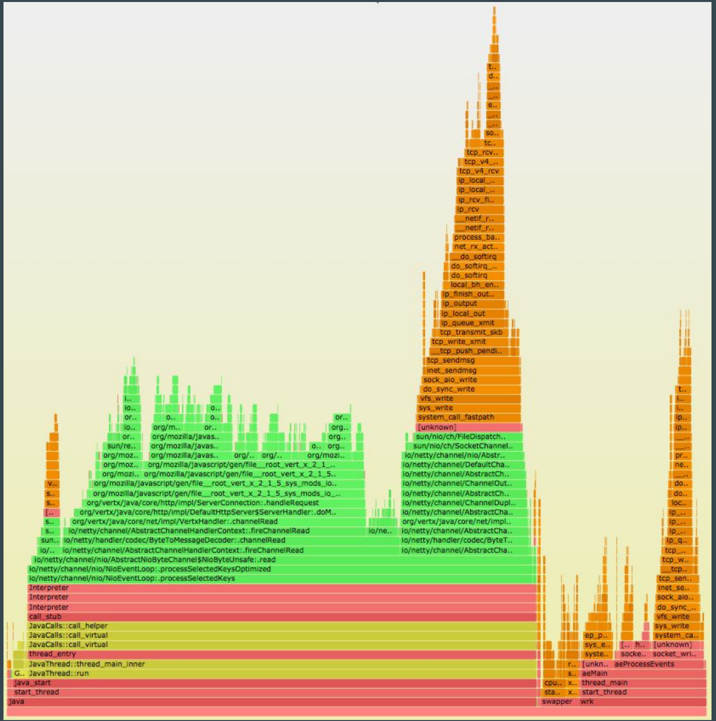
```
$ jstack -l $PID > output
$ kill -3 $PID # goes to stderr, wherever this is directed to
```

- `-l` gives additional info about locks
- https://github.com/spotify/threaddump-analyzer

WARN!

- A thread's stack is only retrieved at safepoints (most JVMs)
  - Hurts accuracy of reported stacks
  - This also affects profilers
- One thread's stack dumped at a time
  - Inconsistent stacks: two threads hold the same lock; thread blocked on free monitor.. (you can see this in a few slides)

FLAME GRAPHS



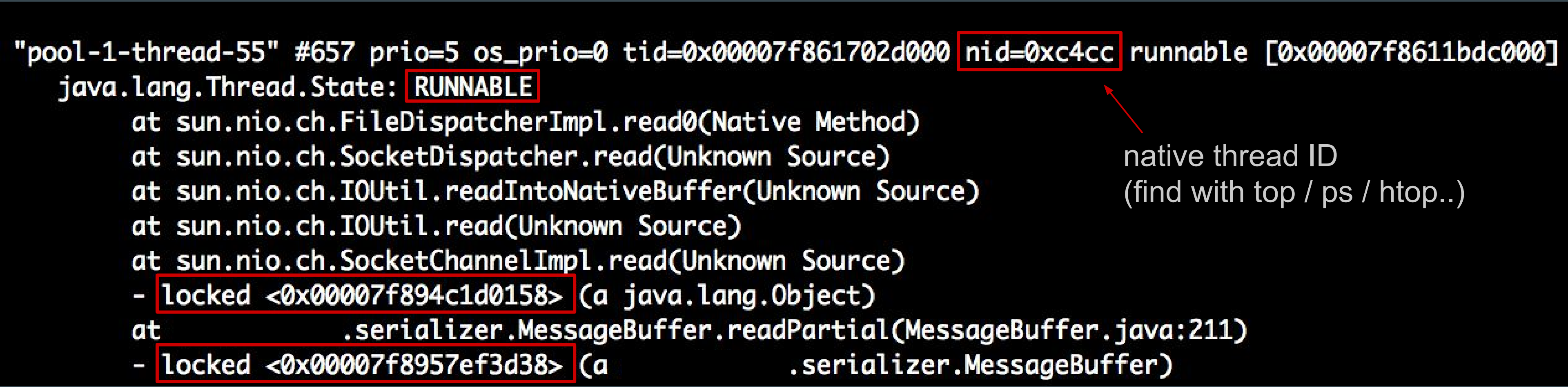
Stack visualization, crossing JVM → OS

Very effective to spot where CPU time is going

Drill down to specific sections of the stack

http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

Synchronization

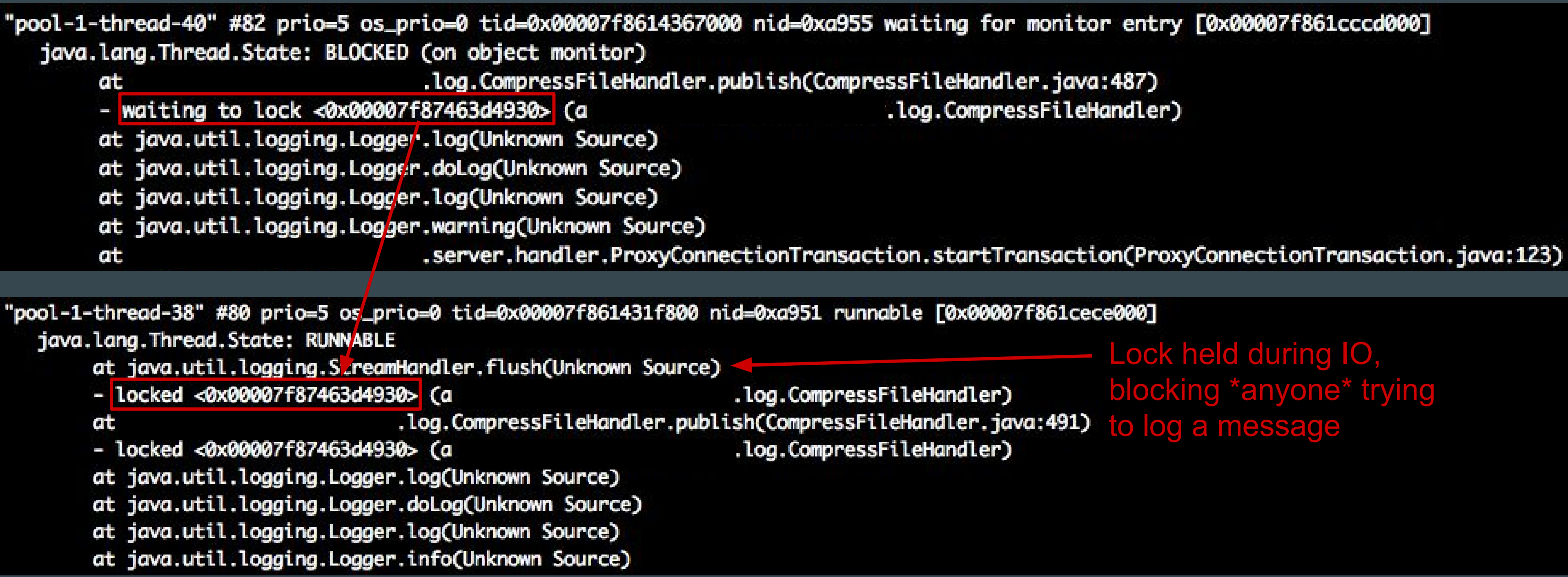


```
"pool-1-thread-55" #657 prio=5 os_prio=0 tid=0x00007f861702d000 nid=0xc4cc runnable [0x00007f8611bdc000]
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.FileDispatcherImpl.read0(Native Method)
        at sun.nio.ch.SocketDispatcher.read(Unknown Source)
        at sun.nio.ch.IOUtil.readIntoNativeBuffer(Unknown Source)
        at sun.nio.ch.IOUtil.read(Unknown Source)
        at sun.nio.ch.SocketChannelImpl.read(Unknown Source)
        - locked <0x00007f894c1d0158> (a java.lang.Object)
        at            .serializer.MessageBuffer.readPartial(MessageBuffer.java:211)
        - locked <0x00007f8957ef3d38> (a            .serializer.MessageBuffer)
```

native thread ID
(find with top / ps / htop..)

Two monitors held while we don’t do anything (waiting for OS on an IO read)

Synchronization



```
"pool-1-thread-40" #82 prio=5 os_prio=0 tid=0x00007f8614367000 nid=0xa955 waiting for monitor entry [0x00007f861cccd000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at                         .log.CompressFileHandler.publish(CompressFileHandler.java:487)
        - waiting to lock <0x00007f87463d4930> (a                                      .log.CompressFileHandler)
        at java.util.logging.Logger.log(Unknown Source)
        at java.util.logging.Logger.doLog(Unknown Source)
        at java.util.logging.Logger.log(Unknown Source)
        at java.util.logging.Logger.warning(Unknown Source)
        at                         .server.handler.ProxyConnectionTransaction.startTransaction(ProxyConnectionTransaction.java:123)
```

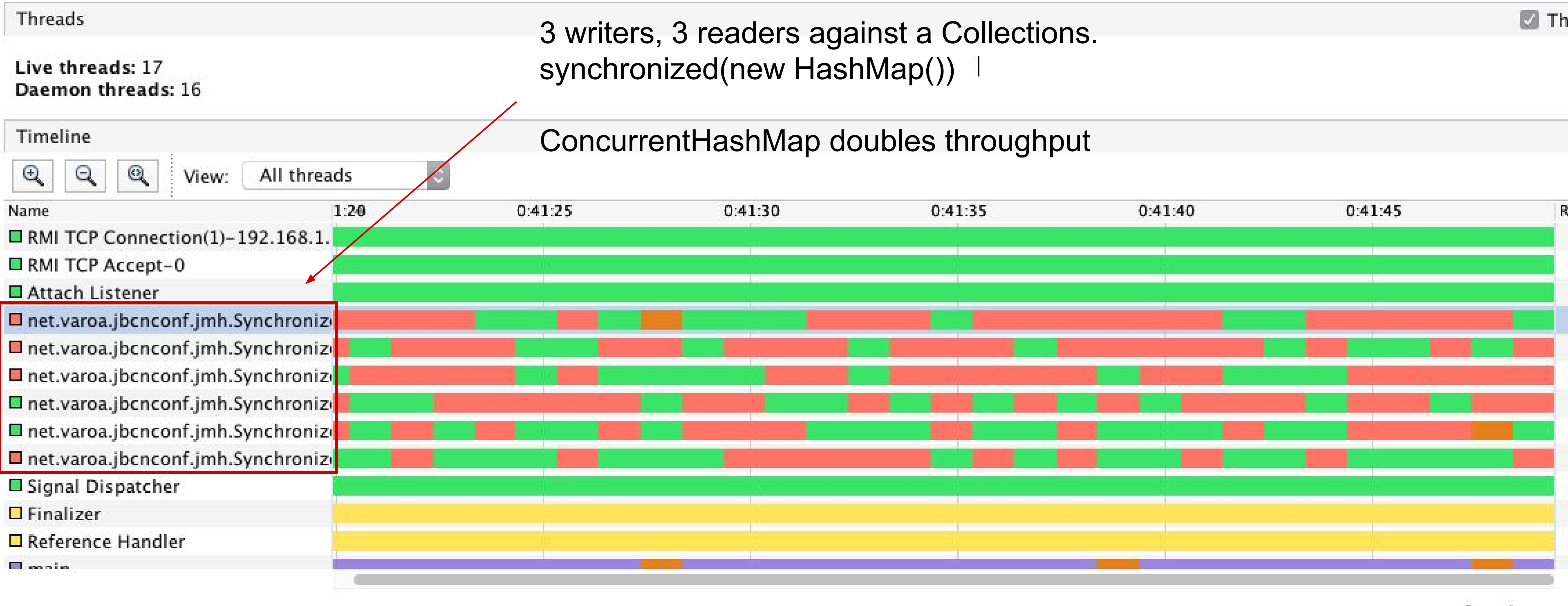
```
"pool-1-thread-38" #80 prio=5 os_prio=0 tid=0x00007f861431f800 nid=0xa951 runnable [0x00007f861cece000]
   java.lang.Thread.State: RUNNABLE
        at java.util.logging.StreamHandler.flush(Unknown Source)
        - locked <0x00007f87463d4930> (a                         .log.CompressFileHandler)
        at                         .log.CompressFileHandler.publish(CompressFileHandler.java:491)
        - locked <0x00007f87463d4930> (a                         .log.CompressFileHandler)
        at java.util.logging.Logger.log(Unknown Source)
        at java.util.logging.Logger.doLog(Unknown Source)
        at java.util.logging.Logger.log(Unknown Source)
        at java.util.logging.Logger.info(Unknown Source)
```

Lock held during IO,
blocking *anyone* trying
to log a message

Synchronization

3 writers, 3 readers against a Collections.
synchronized(new HashMap())

ConcurrentHashMap doubles throughput



Threads

Live threads: 17
Daemon threads: 16

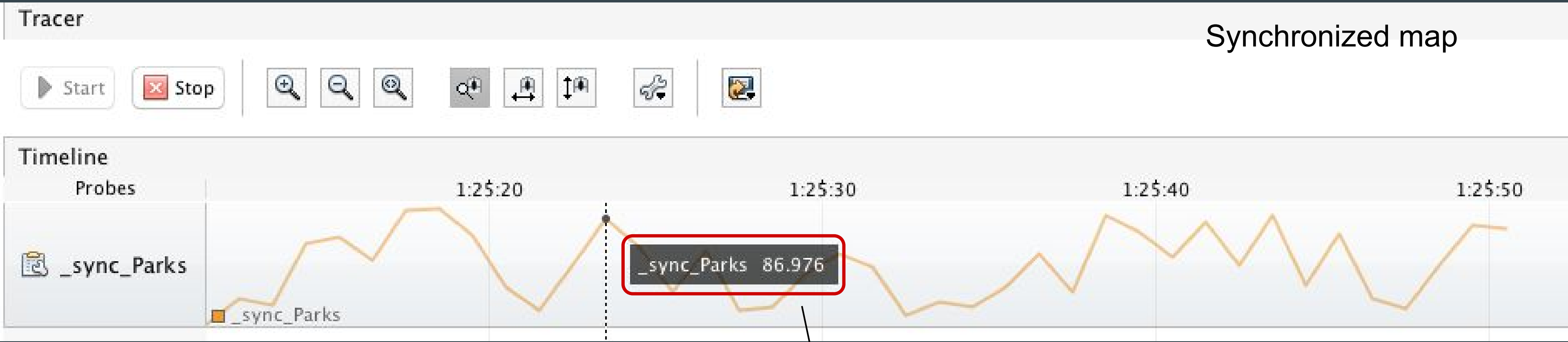
Timeline

View: All threads

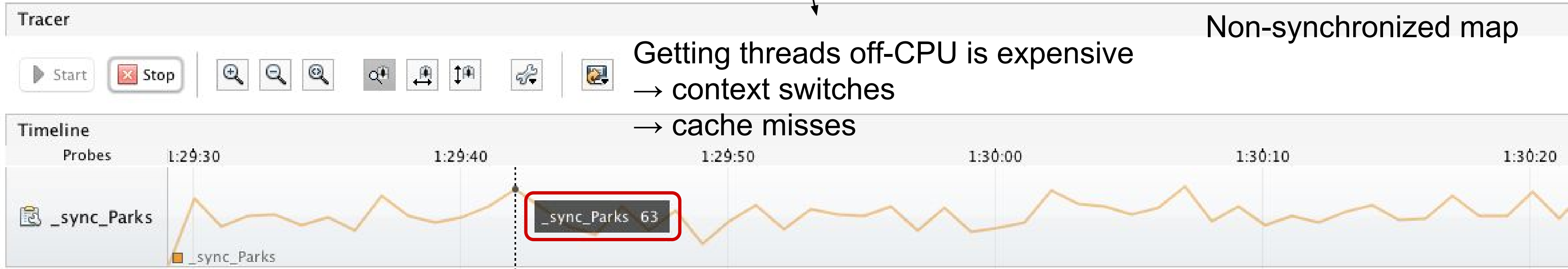
Th
Name
1:20
0:41:25
0:41:30
0:41:35
0:41:40
0:41:45
R
RMI TCP Connection(1)-192.168.1.
RMI TCP Accept-0
Attach Listener
net.varoa.jbcnconf.jmh.Synchroniz
net.varoa.jbcnconf.jmh.Synchroniz
net.varoa.jbcnconf.jmh.Synchroniz
net.varoa.jbcnconf.jmh.Synchroniz
net.varoa.jbcnconf.jmh.Synchroniz
net.varoa.jbcnconf.jmh.Synchroniz
Signal Dispatcher
Finalizer
Reference Handler
main

Synchronization

```
jcmd $PID PerfCounter.print | grep Parks
```



Synchronized map



Non-synchronized map

Getting threads off-CPU is expensive
→ context switches
→ cache misses

Synchronization

Synchronized blocks imply serial parts of the program

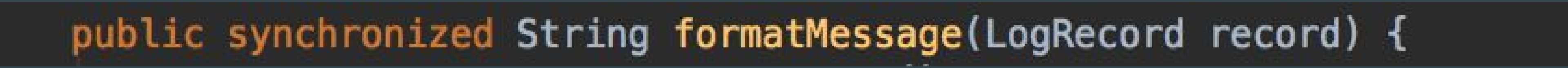
Amhdal's law:

$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

- A x10 speedup of 10% of the exec. time (p = 0.1, s = 10) → ~1.10 speedup
- A x1.5 speedup of 90% of the exec. time (p = 0.9, s = 1.5) → ~1.43 speedup

Avoid

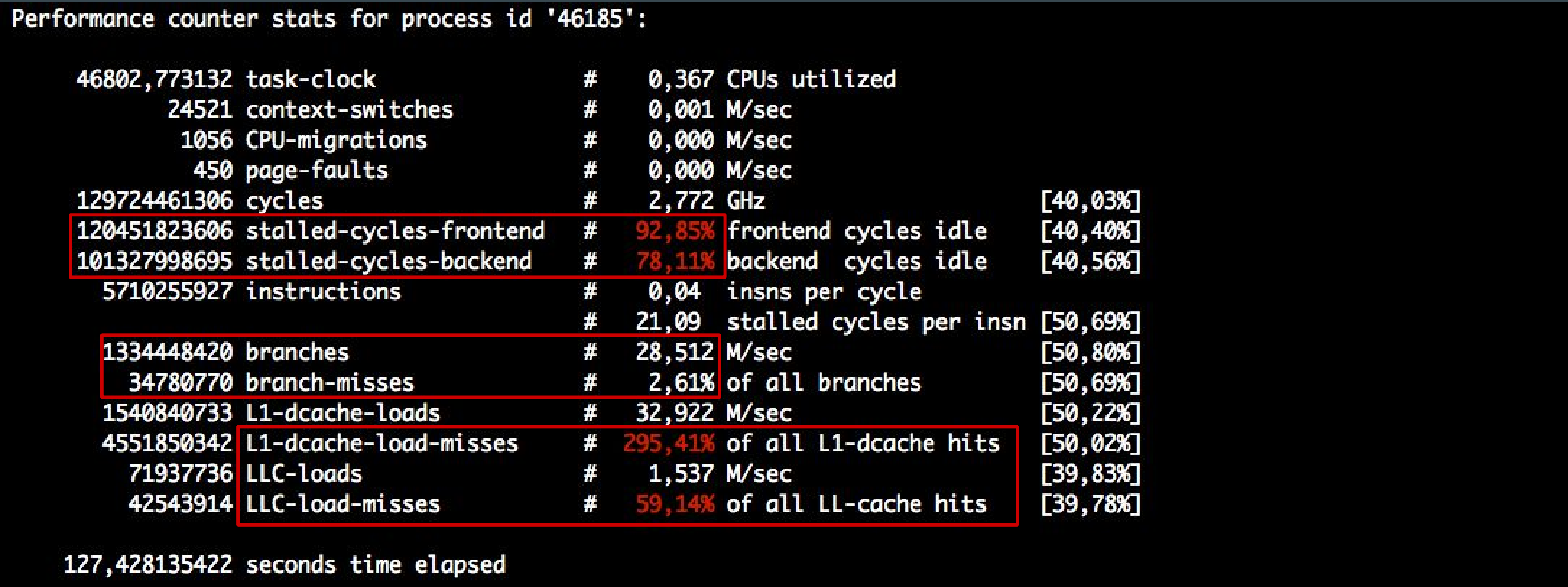
- Large synchronized blocks
- Synchronizing on `this` (you're becoming vulnerable to potential blocks)
- Calling external methods while synchronized (e.g.: see below, in the JDK)



```
public synchronized String formatMessage(LogRecord record) {
```

CPU utilisation (or lack of)

```
perf stat -d -p $PID   # also: cat /proc/$PID/status
```



```
Performance counter stats for process id '46185':

     46802,773132 task-clock                #    0,367 CPUs utilized
            24521 context-switches          #    0,001 M/sec
             1056 CPU-migrations            #    0,000 M/sec
              450 page-faults               #    0,000 M/sec
     129724461306 cycles                    #    2,772 GHz                     [40,03%]
     120451823606 stalled-cycles-frontend   #   92,85% frontend cycles idle    [40,40%]
     101327998695 stalled-cycles-backend    #   78,11% backend  cycles idle    [40,56%]
       5710255927 instructions              #    0,04  insns per cycle
                                            #   21,09  stalled cycles per insn [50,69%]
       1334448420 branches                  #   28,512 M/sec                   [50,80%]
         34780770 branch-misses             #    2,61% of all branches         [50,69%]
       1540840733 L1-dcache-loads           #   32,922 M/sec                   [50,22%]
       4551850342 L1-dcache-load-misses     #  295,41% of all L1-dcache hits   [50,02%]
         71937736 LLC-loads                 #    1,537 M/sec                   [39,83%]
         42543914 LLC-load-misses           #   59,14% of all LL-cache hits    [39,78%]

    127,428135422 seconds time elapsed
```

Concurrency toolbox

The JDK offers resources for concurrency

- `java.util.concurrent.locks.*`
- `java.util.concurrent.* → e.g.: thread safe collections`

Does your application really need locks?

- `volatile` : writes guaranteed to be visible when other threads read
- AtomicXX classes, operations: `incrementAndGet`, `compareAndSwap ...`
- `AtomicXX.lazySet()` : writes not reordered with later writes (cheaper for single writer)

High performance lock-free collections: https://github.com/JCTools/JCTools

Concurrency toolbox: coordination

Example:

```
Queue<T> queue; // unbounded queue, we can’t change the implementation
...
public void onNext(T t) {
     queue.offer(t)                              // enqueues a new item
     pollQueue(wip, requested, queue, child) // drains `requested` items from the queue
}
```

How to bound the queue, and trigger a backpressure notification?

```
synchronized(queue) {
     if (queue.size() < limit)
          queue.offer(t);
     else
          callback.apply()
          return false;
}
```

serial block and risk of parking threads

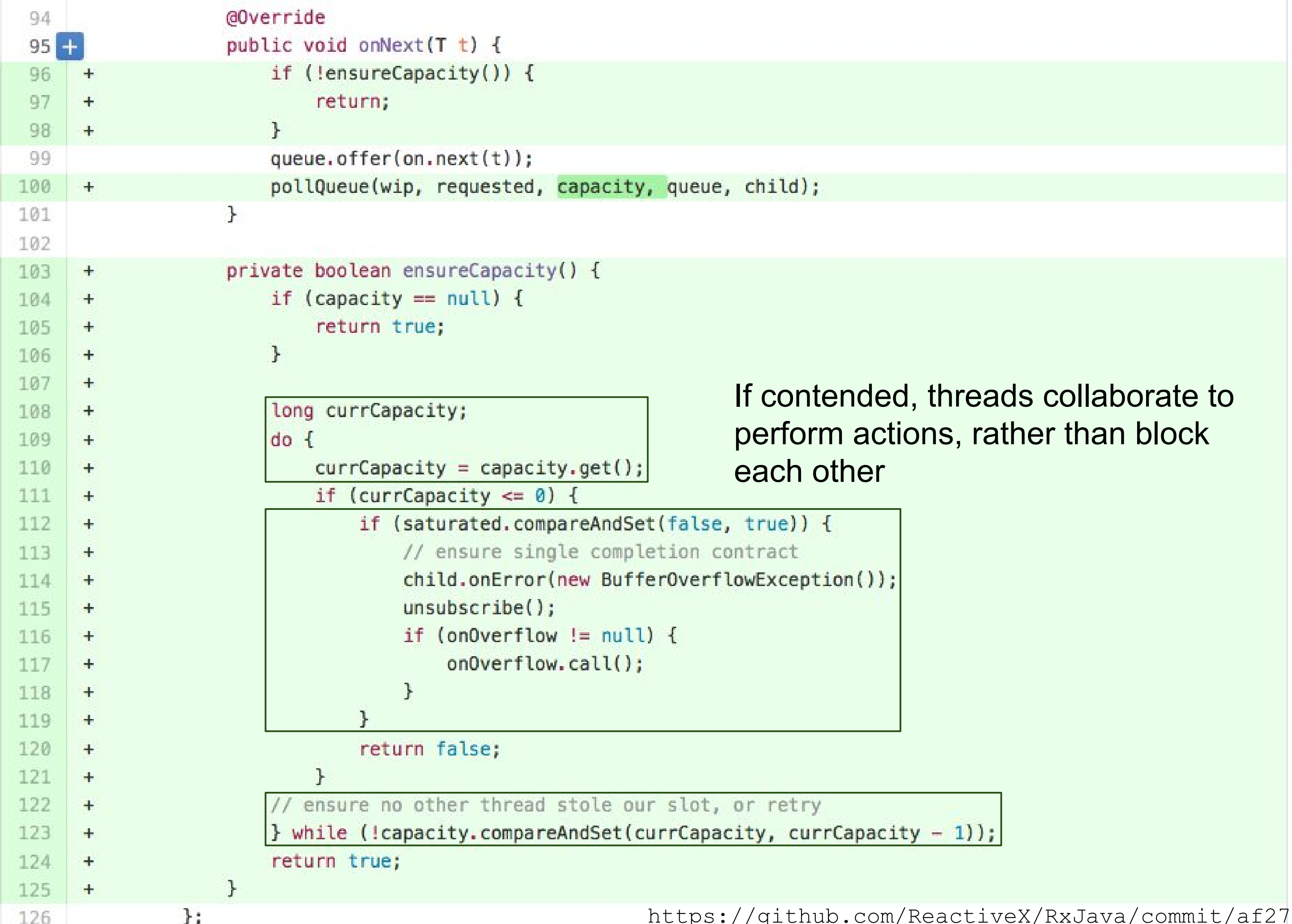
what if this blocks? we’re effectively blocking anyone else accessing the queue, even consumers

AtomicInteger

Snapshot state

If full, make sure exactly 1 thread deals with the consequences

Confirm slot for our item if nothing changed



```
        @Override
        public void onNext(T t) {
+           if (!ensureCapacity()) {
+               return;
+           }
            queue.offer(on.next(t));
+           pollQueue(wip, requested, capacity, queue, child);
        }

+       private boolean ensureCapacity() {
+           if (capacity == null) {
+               return true;
+           }
+
+           long currCapacity;
+           do {
+               currCapacity = capacity.get();
+               if (currCapacity <= 0) {
+                   if (saturated.compareAndSet(false, true)) {
+                       // ensure single completion contract
+                       child.onError(new BufferOverflowException());
+                       unsubscribe();
+                       if (onOverflow != null) {
+                           onOverflow.call();
+                       }
+                   }
+                   return false;
+               }
+           // ensure no other thread stole our slot, or retry
+           } while (!capacity.compareAndSet(currCapacity, currCapacity - 1));
+           return true;
+       }
    };
```

If contended, threads collaborate to perform actions, rather than block each other

https://github.com/ReactiveX/RxJava/commit/af2756

Concurrency toolbox: ThreadLocal

```
class Formatter {
  DateFormat df = new SimpleDateFormat(“DDMMYYYY”)     // SDF is not thread safe ..
  public String format(Date d) {                       // .. so this is neither
    return df.format(d);                               // Can we avoid creating an
  }                                                    // instance per .format() call?
}
```

```
class Formatter {
  ThreadLocal<SimpleDateFormat> df =new ThreadLocal() { // Wrap it in ThreadLocal
     public SimpleDateFormat initialValue() {           // Called on the first .get()
          return new SimpleDateFormat();                // performed by each Thread
     }
  };
  public String format(Date d) {                        // Now thread-safe as each
     return df.get().format(d);                         // thread gets its own instance
  }                                                     // of SimpleDateFormat
}
```

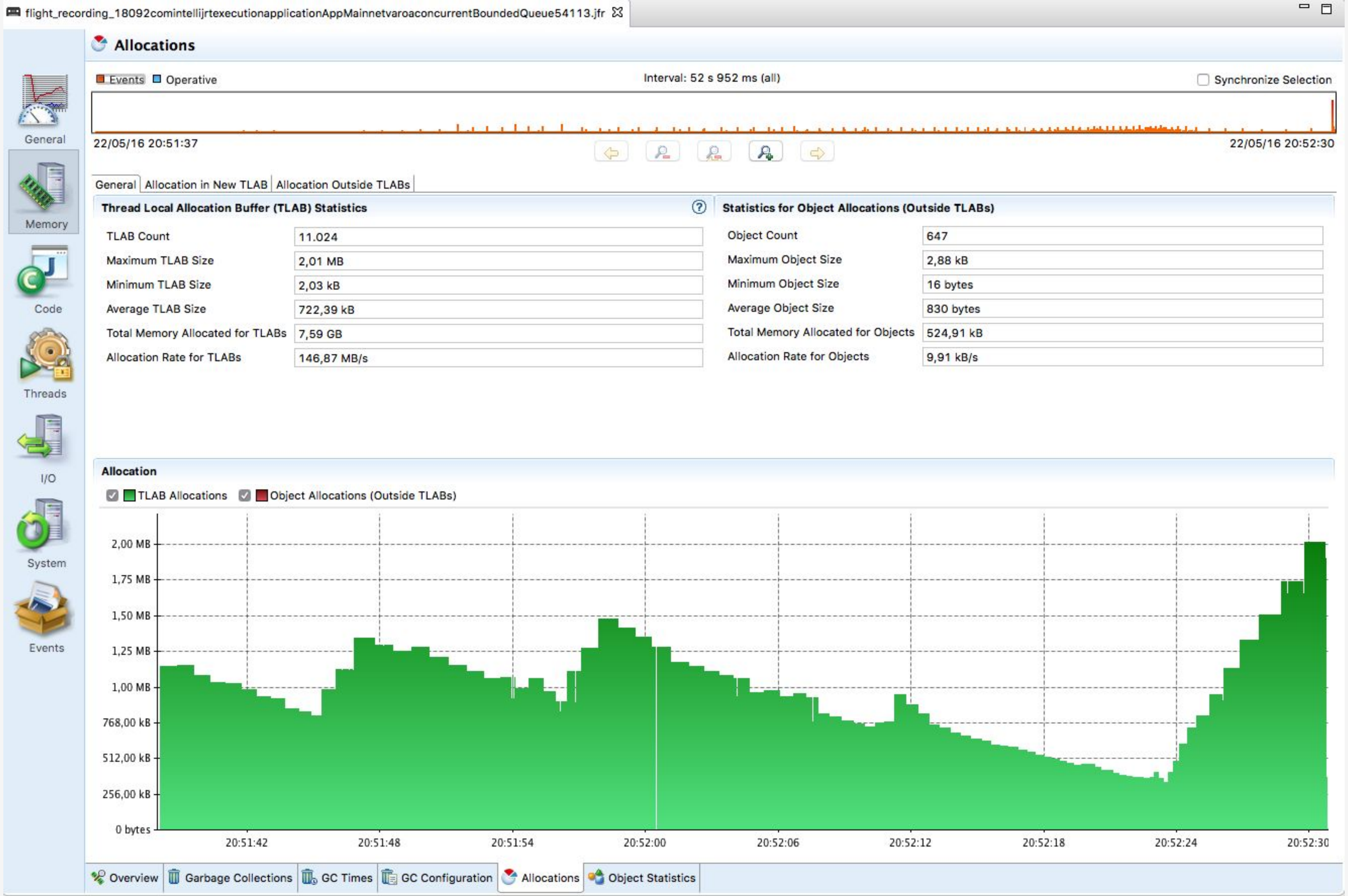
Concurrency toolbox: Thread Local buffers

This is actually how the JVM deals with allocations from multiple threads. Avoids contention by allocating on Thread-Local Allocation Buffers.

- Enabled by default: `-XX:+UseTLAB`
- See details in Flight Recorder (next slide), or: `-XX:+PrintTLAB`

```
TLAB: gc thread: 0x00007f3c1ff0f800 [id: 10519] desired_size: 221KB
     slow allocs: 8  refill waste: 3536B alloc: 0.01613    11058KB
     refills: 73 waste  0.1% gc: 10368B slow: 2112B fast: 0B
```

● `-XX:+ResizeTLAB`	let the JVM dynamically adjust size
● `-XX:TLABSize=2m -XX:MinTLABSize=64k`	adjust it yourself
● `-XX:+AggressiveOpts`	



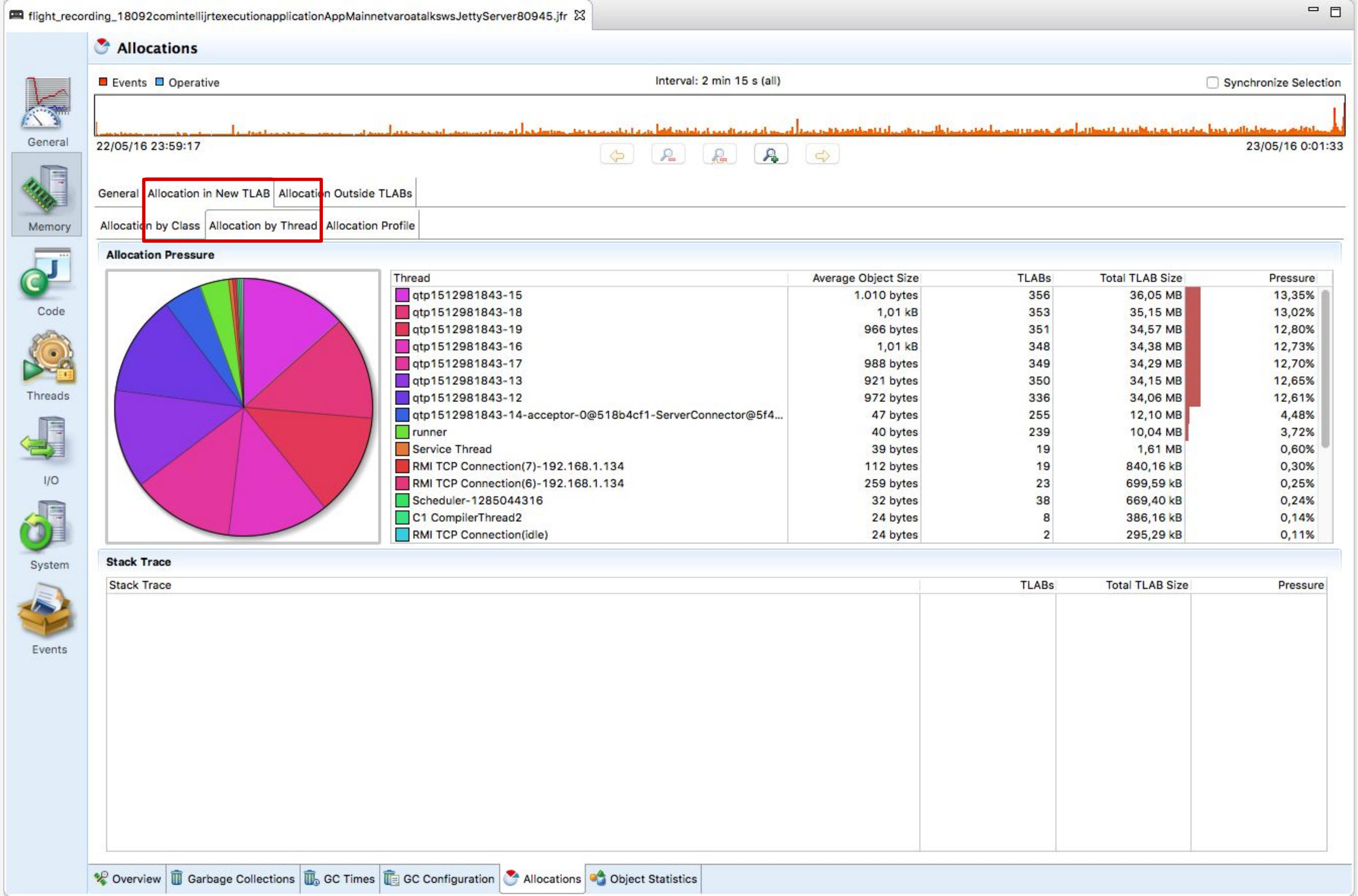
Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	11.024
Maximum TLAB Size	2,01 MB
Minimum TLAB Size	2,03 kB
Average TLAB Size	722,39 kB
Total Memory Allocated for TLABs	7,59 GB
Allocation Rate for TLABs	146,87 MB/s

Statistics for Object Allocations (Outside TLABs)

Object Count	647
Maximum Object Size	2,88 kB
Minimum Object Size	16 bytes
Average Object Size	830 bytes
Total Memory Allocated for Objects	524,91 kB
Allocation Rate for Objects	9,91 kB/s

Allocation



Thread	Average Object Size	TLABs	Total TLAB Size	Pressure
qtp1512981843-15	1.010 bytes	356	36,05 MB	13,35%
qtp1512981843-18	1,01 kB	353	35,15 MB	13,02%
qtp1512981843-19	966 bytes	351	34,57 MB	12,80%
qtp1512981843-16	1,01 kB	348	34,38 MB	12,73%
qtp1512981843-17	988 bytes	349	34,29 MB	12,70%
qtp1512981843-13	921 bytes	350	34,15 MB	12,65%
qtp1512981843-12	972 bytes	336	34,06 MB	12,61%
qtp1512981843-14-acceptor-0@518b4cf1-ServerConnector@5f4...	47 bytes	255	12,10 MB	4,48%
runner	40 bytes	239	10,04 MB	3,72%
Service Thread	39 bytes	19	1,61 MB	0,60%
RMI TCP Connection(7)-192.168.1.134	112 bytes	19	840,16 kB	0,30%
RMI TCP Connection(6)-192.168.1.134	259 bytes	23	699,59 kB	0,25%
Scheduler-1285044316	32 bytes	38	669,40 kB	0,24%
C1 CompilerThread2	24 bytes	8	386,16 kB	0,14%
RMI TCP Connection(idle)	24 bytes	2	295,29 kB	0,11%

Stack Trace	TLABs	Total TLAB Size	Pressure

Unintended contention

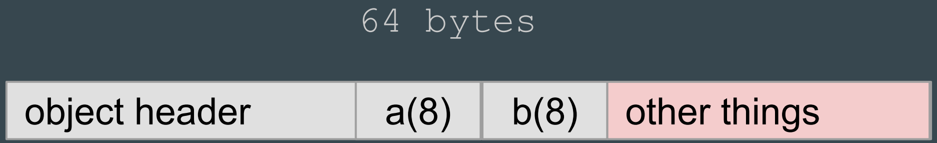
False sharing

```
class MyClass[T] {
    public volatile long a = 0;
    public volatile long b = 0;
}
```

Thread A writes to `a` in CPU1

Thread B reads `b` in CPU2

Do we have contention? Yes



64 bytes

object header | a(8) | b(8) | other things

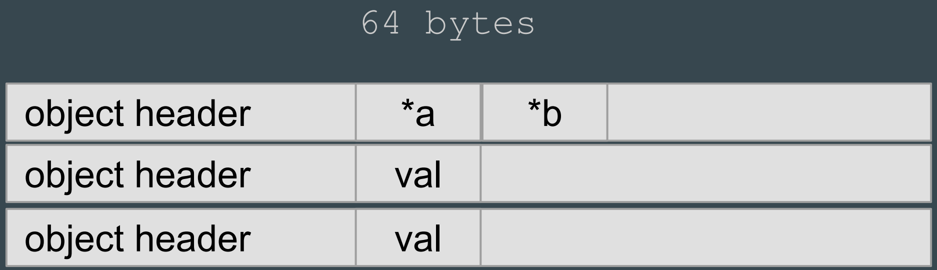
The CPU works with a full cache line, not individual fields.

A takes exclusive ownership of the full cache line, updates and B’s copy is invalidated, even if B’s value didn’t change. Same applies to the rest of the line

Unintended contention

False sharing

```
class MyClass[T] {
    PaddedAtomicLong a;
    PaddedAtomicLong b;
}

class PaddedAtomicLong extends AtomicLong {
    public final long
        p1, p2, p3, p4, p5, p6 = 7L;
}
```



64 bytes

object header	*a	*b	
object header	val		
object header	val		

Padding ensures they reside on different cache lines

http://mechanical-sympathy.blogspot.com.es/2011/07/false-sharing.html
http://mechanical-sympathy.blogspot.com.es/2011/08/false-sharing-java-7.html
http://psy-lob-saw.blogspot.com.es/2014/03/java-object-layout-tale-of-confusion.html

Leftovers

- Java Microbenchmark Harness (JMH)
- Have performance targets as business requirements
- Perf tests are hard (e.g.: generating more load than production)
- Instrumentation, monitorization
    - As part of CI: add performance tests early, check for regressions...
    - Production: make behaviour visible, spot anomalies
- No need to optimize early, but have a story for how you’d improve when needed

Q & A

Thanks!